A Reduction Theorem for Store Buffers

Ernie Cohen\textsuperscript{1}, Norbert Schirmer\textsuperscript{2},\textsuperscript{*}

\textsuperscript{1} Microsoft Corp., Redmond, WA, USA
\textsuperscript{2} German Research Center for Artificial Intelligence (DFKI) Saarbrücken, Germany
ecohen@amazon.com, norbert.schirmer@web.de

Abstract. When verifying a concurrent program, it is usual to assume that memory is sequentially consistent. However, most modern multiprocessors depend on store buffering for efficiency, and provide native sequential consistency only at a substantial performance penalty. To regain sequential consistency, a programmer has to follow an appropriate programming discipline. However, naïve disciplines, such as protecting all shared accesses with locks, are not flexible enough for building high-performance multiprocessor software.

We present a new discipline for concurrent programming under TSO (total store order, with store buffer forwarding). It does not depend on concurrency primitives, such as locks. Instead, threads use ghost operations to acquire and release ownership of memory addresses. A thread can write to an address only if no other thread owns it, and can read from an address only if it owns it or it is shared and the thread has flushed its store buffer since it last wrote to an address it did not own. This discipline covers both coarse-grained concurrency (where data is protected by locks) as well as fine-grained concurrency (where atomic operations race to memory).

We formalize this discipline in Isabelle/HOL, and prove that if every execution of a program in a system without store buffers follows the discipline, then every execution of the program with store buffers is sequentially consistent. Thus, we can show sequential consistency under TSO by ordinary assertional reasoning about the program, without having to consider store buffers at all.

\textsuperscript{*} Work funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.
1 Introduction

When verifying a shared-memory concurrent program, it is usual to assume that each memory operation works directly on a shared memory state, a model sometimes called atomic memory. A memory implementation that provides this abstraction for programs that communicate only through shared memory is said to be sequentially consistent. Concurrent algorithms in the computing literature tacitly assume sequential consistency, as do most application programmers.

However, modern computing platforms typically do not guarantee sequential consistency for arbitrary programs, for two reasons. First, optimizing compilers are typically incorrect unless the program is appropriately annotated to indicate which program locations might be concurrently accessed by other threads; this issue is addressed only cursorily in this report. Second, modern processors buffer stores of retired instructions. To make such buffering transparent to single-processor programs, subsequent reads of the processor read from these buffers in preference to the cache. (Otherwise, a program could write a new value to an address but later read an older value.) However, in a multiprocessor system, processors do not snoop the store buffers of other processors, so a store is visible to the storing processor before it is visible to other processors. This can result in executions that are not sequentially consistent.
The simplest example illustrating such an inconsistency is the following program, consisting of two threads T0 and T1, where x and y are shared memory variables (initially 0) and r0 and r1 are registers:

\[
\begin{align*}
T0 & \quad T1 \\
& \quad \begin{align*}
x & = 1; \quad y = 1; \\
r0 & = y; \quad r1 = x;
\end{align*}
\end{align*}
\]

In a sequentially consistent execution, it is impossible for both r0 and r1 to be assigned 0. This is because the assignments to x and y must be executed in some order; if x (resp. y) is assigned first, then r1 (resp. r0) will be set to 1. However, in the presence of store buffers, the assignments to r0 and r1 might be performed while the writes to x and y are still in their respective store buffers, resulting in both r0 and r1 being assigned 0.

One way to cope with store buffers is make them an explicit part of the programming model. However, this is a substantial programming concession. First, because store buffers are FIFO, it ratchets up the complexity of program reasoning considerably; for example, the reachability problem for a finite set of concurrent finite-state programs over a finite set of finite-valued locations is in PSPACE without store buffers, but undecidable (even for two threads) with store buffers. Second, because writes from function calls might still be buffered when a function returns, making the store buffers explicit would break modular program reasoning.

In practice, the usual remedy for store buffering is adherence to a programming discipline that provides sequential consistency for a suitable class of architectures. In this report, we describe and prove the correctness of such a discipline suitable for the memory model provided by existing x86/x64 machines, where each write emerging from a store buffer hits a global cache visible to all processors. Because each processor sees the same global ordering of writes, this model is sometimes called total store order (TSO) [2].

The concurrency discipline most familiar to concurrent programs is one where each variable is protected by a lock, and a thread must hold the corresponding lock to access the variable. (It is possible to generalize this to allow shared locks, as well as variants such as split semaphores.) Such lock-based techniques are typically referred to as coarse-grained concurrency control, and suffice for most concurrent application programming. However, these techniques do not suffice for low-level system programming (e.g., the construction of OS kernels), for several reasons. First, in kernel programming efficiency is paramount, and atomic memory operations are more efficient for many problems. Second, lock-free concurrency control can sometimes guarantee stronger correctness (e.g., wait-free algorithms can provide bounds on execution time). Third, kernel programming requires taking into account the implicit concurrency of concurrent hardware activities (e.g., a hardware TLB racing to use page tables while the kernel is trying to access them), and hardware cannot be forced to follow a locking discipline.

A more refined concurrency control discipline, one that is much closer to expert practice, is to classify memory addresses as lock-protected or shared. Lock-protected addresses are used in the usual way, but shared addresses can be accessed using atomic operations provided by hardware (e.g., on x86 class architectures, most reads and writes are atomic). The main restriction on these accesses is that if a processor does a shared write and a

---

3 Before 2008, Intel [9] and AMD [1] both put forward a weaker memory model in which writes to different memory addresses may be seen in different orders on different processors, but respecting causal ordering. However, current implementations satisfy the stronger conditions described in this report and are also compliant with the latest revisions of the Intel specifications [10]. According to Owens et al. [15] AMD is also planning a similar adaptation of their manuals.

4 This atomicity isn’t guaranteed for certain memory types, or for operations that cross a cache line.
subsequent shared read (possibly from a different address), the processor must flush the store buffer somewhere in between. For example, in the example above, both \( x \) and \( y \) would be shared addresses, so each processor would have to flush its store buffer between its first and second operations.

However, even this discipline is not very satisfactory. First, we would need even more rules to allow locks to be created or destroyed, or to change memory between shared and protected, and so on. Second, there are many interesting concurrency control primitives, and many algorithms, that allow a thread to obtain exclusive ownership of a memory address; why should we treat locking as special?

In this report, we consider a much more general and powerful discipline that also guarantees sequential consistency. The basic rule for shared addresses is similar to the discipline above, but there are no locking primitives. Instead, we treat ownership as fundamental. The difference is that ownership is manipulated by nonblocking ghost updates, rather than an operation like locking that have runtime overhead. Informally the rules of the discipline are as follows:

- In any state, each memory address is either shared or unshared. Each memory address is also either owned by a unique thread or unowned. Every unowned address must be shared. Each address is also either read-only or read-write. Every read-only address is unowned.
- A thread can (autonomously) acquire ownership of an unowned address, or release ownership of a address that it owns. It can also change whether an address it owns is shared or not. Upon release of an address it can mark it as read-only.
- Each memory access is marked as volatile or non-volatile.
- A thread can perform a write if it is sound. It can perform a read if it is sound and clean.
- A non-volatile write is sound if the thread owns the address and the address is unshared.
- A non-volatile read is sound if the thread owns the address or the address is read-only.
- A volatile write is sound if no other thread owns the address and the address is not marked as read-only.
- A volatile read is sound if the address is shared or the thread owns it.
- A volatile read is clean if the store buffer has been flushed since the last volatile write. Moreover, every non-volatile read is clean.
- For interlocked operations (like compare and swap), which have the side effect of the store buffer getting flushed, the rules for volatile accesses apply.

Note first that these conditions are not thread-local, because some actions are allowed only when an address is unowned, marked read-only, or not marked read-only. A thread can ascertain such conditions only through system-wide invariants, respected by all threads, along with data it reads. By imposing suitable global invariants, various thread-local disciplines (such as one where addresses are protected by locks, conditional critical reasons, or monitors) can be derived as lemmas by ordinary program reasoning, without need for meta-theory.

Second, note that these rules can be checked in the context of a concurrent program without store buffers, by introducing ghost state to keep track of ownership and sharing and whether the thread has performed a volatile write since the last flush. Our main result is that if a program obeys the rules above, then the program is sequentially consistent when executed on a TSO machine.

Consider our first example program. If we choose to leave both \( x \) and \( y \) unowned (and hence shared), then all accesses must be volatile. This would force each thread to flush the store buffer between their first and second operations. In practice, on an x86/x64 machine,
this would be done by making the writes interlocked, which flushes store buffers as a side effect. Whichever thread flushes its store buffer second is guaranteed to see the write of the other thread, making the execution violating sequential consistency impossible.

However, couldn’t the first thread try to take ownership of \( x \) before writing it, so that its write could be non-volatile? The answer is that it could, but then the second thread would be unable to read \( x \) volatile (or take ownership of \( x \) and read it non-volatile), because we would be unable to prove that \( x \) is unowned at that point. In other words, a thread can take ownership of an address only if it is not racing to do so.

Ultimately, the races allowed by the discipline involve volatile access to a shared address, which brings us back to locks. A spinlock is typically implemented with an interlocked read-modify-write on an address (the interlocking providing the required flushing of the store buffer). If the locking succeeds, we can prove (using for example a ghost variable giving the ID of the thread taking the lock) that no other thread holds the lock, and can therefore safely take ownership of an address “protected” by the lock (using the global invariant that only the lock owner can own the protected address). Thus, our discipline subsumes the better-known disciplines governing coarse-grained concurrency control.

To summarize, our motivations for using ownership as our core notion of a practical programming discipline are the following:

1. the distinction between global (volatile) and local (non-volatile) accesses is a practical requirement to reduce the performance penalty due to necessary flushes and to allow important compiler optimizations (such as moving a local write ahead of a global read),
2. coarse-grained concurrency control like locking is nothing special but only a derived concept which is used for ownership transfer (any other concurrency control that guarantees exclusive access is also fine), and
3. we want that the conditions to check for the programming discipline can be discharged by ordinary state-based program reasoning on a sequentially consistent memory model (without having to talk about histories or complete executions).

Overview In Section 2 we introduce preliminaries of Isabelle/HOL, the theorem prover in which we mechanized our work. In Section 3 we informally describe the programming discipline and basic ideas of the formalization, which is detailed in Section 4 where we introduce the formal models and the reduction theorem. In Section 5 we give some details of important building blocks for the proof of the reduction theorem. To illustrate the connection between a programming language semantics and our reduction theorem, we instantiate our framework with a simple semantics for a parallel WHILE language in Section 6. Finally we conclude in Section 7.

2 Preliminaries

The formalization presented in this papaer is mechanized and checked within the generic interactive theorem prover Isabelle [16]. Isabelle is called generic as it provides a framework to formalize various object logics declared via natural deduction style inference rules. The object logic that we employ for our formalization is the higher order logic of Isabelle/HOL [12].

This article is written using Isabelle’s document generation facilities, which guarantees a close correspondence between the presentation and the actual theory files. We distinguish formal entities typographically from other text. We use a sans serif font for types and constants (including functions and predicates), e.g., \texttt{map}, a slanted serif font for free variables, e.g., \( x \), and a slanted sans serif font for bound variables, e.g., \( x \). Small capitals
are used for data type constructors, e.g., Foo, and type variables have a leading tick, e.g., ′a. HOL keywords are typeset in type-writer font, e.g., let.

To group common premises and to support modular reasoning Isabelle provides locales [4, 5]. A locale provides a name for a context of fixed parameters and premises, together with an elaborate infrastructure to define new locales by inheriting and extending other locales, prove theorems within locales and interpret (instantiate) locales. In our formalization we employ this infrastructure to separate the memory system from the programming language semantics.

The logical and mathematical notions follow the standard notational conventions with a bias towards functional programming. We only present the more unconventional parts here. We prefer curried function application, e.g., \( f \ a \ b \) instead of \( f(a, b) \). In this setting the latter becomes a function application to one argument, which happens to be a pair.

Isabelle/HOL provides a library of standard types like Booleans, natural numbers, integers, total functions, pairs, lists, and sets. Moreover, there are packages to define new data types and records. Isabelle allows polymorphic types, e.g., ′a list is the list type with type variable ′a. In HOL all functions are total, e.g., nat ⇒ nat is a total function on natural numbers. A function update is \( f(y := v) = (\lambda x. \text{if } x = y \text{ then } v \text{ else } f(x)) \). To formalize partial functions the type ′a option is used. It is a data type with two constructors, one to inject values of the base type, e.g., \( \lfloor x \rfloor \), and the additional element ⊥. A base value can be projected with the function the, which is defined by the sole equation the \( \lfloor x \rfloor = x \). Since HOL is a total logic the term the ⊥ is still a well-defined yet un(der)specified value. Partial functions are usually represented by the type ′a ⇒ ′b option, abbreviated as ′a → ′b. They are commonly used as maps. We denote the domain of map \( m \) by \( \text{dom } m \). A map update is written as \( m(a \mapsto v) \). We can restrict the domain of a map \( m \) to a set \( A \) by \( m|_A \).

The syntax and the operations for lists are similar to functional programming languages like ML or Haskell. The empty list is [], with \( x \# xs \) the element \( x \) is ‘consed’ to the list \( xs \). With \( xs @ ys \) list \( ys \) is appended to list \( xs \). With the term \( \text{map } f xs \) the function \( f \) is applied to all elements in \( xs \). The length of a list is \( |xs| \), the \( n \)-th element of a list can be selected with \( xs[n] \) and can be updated via \( xs[n := v] \). With \( \text{dropWhile } P xs \) the prefix for which all elements satisfy predicate \( P \) are dropped from list \( xs \).

Sets come along with the standard operations like union, i.e., \( A \cup B \), membership, i.e., \( x \in A \) and set inversion, i.e., \( \neg A \).

Tuples with more than two components are pairs nested to the right.

3 Programming discipline

For sequential code on a single processor the store buffer is invisible, since reads respect outstanding writes in the buffer. This argument can be extended to thread local memory in the context of a multiprocessor architecture. Memory typically becomes temporarily thread local by means of locking. The C-idiom to identify shared portions of the memory is the \texttt{volatile} tag on variables and type declarations. Thread local memory can be accessed non-volatilely, whereas accesses to shared memory are tagged as volatile. This prevents the compiler from applying certain optimizations to those accesses which could cause undesired behavior, e.g., to store intermediate values in registers instead of writing them to the memory.

The basic idea behind the programming discipline is, that before gathering new information about the shared state (via reading) the thread has to make its outstanding changes to the shared state visible to others (by flushing the store buffer). This allows to sequentialize the reads and writes to obtain a sequentially consistent execution of the global system. In this sequentialization a write to shared memory happens when the write
instruction exits the store buffer, and a read from the shared memory happens when all preceding writes have exited.

We distinguish thread local and shared memory by an ownership model. Ownership is maintained in ghost state and can be transferred as side effect of write operations and by a dedicated ghost operation. Every thread has a set of owned addresses. Owned addresses of different threads are disjoint. Moreover, there is a global set of shared addresses which can additionally be marked as read-only. Unowned addresses — addresses owned by no thread — can be accessed concurrently by all threads. They are a subset of the shared addresses. The read-only addresses are a subset of the unowned addresses (and thus of the shared addresses). We only allow a thread to write to owned addresses and unowned, read-write addresses. We only allow a thread to read from owned addresses and from shared addresses (even if they are owned by another thread).

All writes to shared memory have to be volatile. Reads from shared addresses also have to be volatile, except if the address is owned (i.e., single writer, multiple readers) or if the address is read-only. Moreover, non-volatile writes are restricted to owned, unshared memory. As long as a thread owns an address it is guaranteed that it is the only one writing to that address. Hence this thread can safely perform non-volatile reads to that address without missing any write. Similar it is safe for any thread to access read-only memory via non-volatile reads since there are no outstanding writes at all.

Recall that a volatile read is clean if it is guaranteed that there is no outstanding volatile write (to any address) in the store buffer. Moreover every non-volatile read is clean. To regain sequential consistency under the presence of store buffers every thread has to make sure that every read is clean, by flushing the store buffer when necessary. To check the flushing policy of a thread, we keep track of clean reads by means of ghost state. For every thread we maintain a dirty flag. It is reset as the store buffer gets flushed. Upon a volatile write the dirty flag is set. The dirty flag is considered to guarantee that a volatile read is clean.

Table 1a summarizes the access policy and Table 1b the associated flushing policy of the programming discipline. The key motivation is to improve performance by minimizing the number of store buffer flushes, while staying sequentially consistent. The need for flushing the store buffer decreases from interlocked accesses (where flushing is a side-effect) over volatile accesses to non-volatile accesses. From the viewpoint of access rights there is no difference between interlocked and volatile accesses. However, keep in mind that some interlocked operations can read from, modify and write to an address in a single atomic step of the underlying hardware and are typically used in lock-free algorithms or for the implementation of locks.

<table>
<thead>
<tr>
<th>Table 1: Programming discipline.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Access policy</td>
</tr>
<tr>
<td>shared (read-write)</td>
</tr>
<tr>
<td>unowned vR, vW</td>
</tr>
<tr>
<td>owned vR, vW, R</td>
</tr>
<tr>
<td>owned by other vR</td>
</tr>
<tr>
<td>(v)olatile, (R)ead, (W)rite</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Flushing policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>flush (before)</td>
</tr>
<tr>
<td>interlocked as side effect</td>
</tr>
<tr>
<td>vR if not clean</td>
</tr>
<tr>
<td>R, vW, W never</td>
</tr>
</tbody>
</table>
4 Formalization

In this section we go into the details of our formalization. In our model, we distinguish the plain ‘memory system’ from the ‘programming language semantics’ which we both describe as a small-step transition relation. During a computation the programming language issues memory instructions (read / write) to the memory system, which itself returns the results in temporary registers. This clean interface allows us to parameterize the program semantics over the memory system. Our main theorem allows us to simulate a computation step in the semantics based on a memory system with store buffers by $n$ steps in the semantics based on a sequentially consistent memory system. We refer to the former one as store buffer machine and to the latter one as virtual machine. The simulation theorem is independent of the programming language.

We continue with introducing the common parts of both machines. In Section 4.1 we describe the store buffer machine and in Section 4.2 we then describe the virtual machine. The main reduction theorem is presented in 4.3.

Addresses $a$, values $v$ and temporaries $t$ are natural numbers. Ghost annotations for manipulating the ownership information are the following sets of addresses: the acquired addresses $A$, the unshared (local) fraction $L$ of the acquired addresses, the released addresses $R$ and the writable fraction $W$ of the released addresses (the remaining addresses are considered read-only). These ownership annotations are considered as side-effects on volatile writes and interlocked operations (in case a write is performed). Moreover, a special ghost instruction allows to transfer ownership. The possible status changes of an address due to these ownership transfer operations are depicted in Figure 1. Note that ownership of an address is not directly transferred between threads, but is first released by one thread and then can be acquired by another thread. A memory instruction is a datatype with the following constructors:

- **Read volatile** $a \, t$ for reading from address $a$ to temporary $t$, where the Boolean volatile determines whether the access is volatile or not.

- **Write volatile** $a \, sop \, A \, L \, R \, W$ to write the result of evaluating the store operation $sop$ at address $a$. A store operation is a pair $(D, f)$, with the domain $D$ and the function $f$. The function $f$ takes temporaries $\theta$ as a parameter, which maps a temporary to a value. The subset of temporaries that is considered by function $f$ is specified by the domain $D$. We consider store operations as valid when they only depend on their domain:

\[
\text{valid-sop} \, sop \equiv \forall D \, f \, \theta. \, sop = (D, f) \land D \subseteq \text{dom} \, \theta \rightarrow f \, \theta = f \,(\theta|_D)
\]
Again the Boolean volatile specifies the kind of memory access.

- RMW a t sop cond ret A L R W, for atomic interlocked ‘read-modify-write’ instructions (flushing the store buffer). First the value at address a is loaded to temporary t, and then the condition cond on the temporaries is considered to decide whether a store operation is also executed. In case of a store the function ret, depending on both the old value at address a and the new value (according to store operation sop), specifies the final result stored in temporary t. With a trivial condition cond this instruction also covers interlocked reads and writes.

- FENCE, a memory fence that flushes the store buffer.
- GHOST A L R W for ownership transfer.

### 4.1 Store buffer machine

For the store buffer machine the configuration of a single thread is a tuple \((p, is, \vartheta, sb)\) consisting of the program state \(p\), a memory instruction list \(is\), the map of temporaries \(\vartheta\) and the store buffer \(sb\). A global configuration of the store buffer machine \((ts, m)\) consists of a list of thread configurations \(ts\) and the memory \(m\), which is a function from addresses to values.

We describe the computation of the global system by the non-deterministic transition relation \(\stackrel{sb}{\rightarrow} (ts', m')\) defined in Figure 2.

\[
\begin{align*}
&\text{for } i < |ts| \quad ts[i] = (p, is, \vartheta, sb) \\
&\quad \vartheta \vdash p \rightarrow_p (p', is') \\
&\quad (ts, m) \stackrel{sb}{\rightarrow} (ts[i := (p', is', \vartheta, sb)], m) \\
&\text{for } i < |ts| \quad ts[i] = (p, is, \vartheta, sb) \\
&\quad (is, \vartheta, sb, m) \stackrel{m}{\rightarrow} (is', \vartheta', sb', m') \\
&\quad (ts, m) \stackrel{sb}{\rightarrow} (ts[i := (is', \vartheta', sb')], m') \\
&\text{for } i < |ts| \quad ts[i] = (p, is, \vartheta, sb) \\
&\quad (m, sb) \rightarrow_{sb} (m', sb') \\
&\quad (ts, m) \stackrel{sb}{\rightarrow} (ts[i := (p, is, \vartheta, sb')], m')
\end{align*}
\]

Fig. 2: Global transitions of store buffer machine

A transition selects a thread \(ts[i] = (p, is, \vartheta, sb)\) and either the ‘program’ the ‘memory’ or the ‘store buffer’ makes a step defined by sub-relations.

The program step relation is a parameter to the global transition relation. A program step \(\vartheta \vdash p \rightarrow_p (p', is')\) takes the temporaries \(\vartheta\) and the current program state \(p\) and makes a step by returning a new program state \(p'\) and an instruction list \(is'\) which is appended to the remaining instructions.

A memory step \((is, \vartheta, sb, m) \rightarrow m (is', \vartheta', sb', m')\) of a machine with store buffer may only fill its store buffer with new writes.

In a store buffer step \((m, sb) \rightarrow_{sb} (m', sb')\) the store buffer may release outstanding writes to the memory.

The store buffer maintains the list of outstanding memory writes. Write instructions are appended to the end of the store buffer and emerge to memory from the front of the list. A read instructions is satisfied from the store buffer if possible. An entry in the store buffer is of the form WRITE\(_{sb}\) volatile a sop v for an outstanding write (keeping the volatile flag), where operation sop evaluated to value v.

As defined in Figure 3 a write updates the memory when it exits the store buffer.
\((m, \text{Write}_{sb} \text{ volatile} a \ \text{sop} \ v \ A \ L \ R \ W \ # \ sb) \rightarrow_{sb} (m(a := v), sb)\)

Fig. 3: Store buffer transition

\[\begin{align*}
\text{v} &= (\text{case buffered-val sb a of } \perp \Rightarrow m \ a | [\nu'] \Rightarrow \nu') \\
(\text{Read volatile a t # is, } \varnothing, sb, m) &\rightarrow_m (is, \vartheta(t \mapsto v), sb, m) \\
\text{sb}' &= sb \odot [\text{Write}_{sb} \text{ volatile a (D, f) (f } \varnothing \text{) A L R W}] \\
(\text{Write volatile a (D, f) A L R W # is, } \varnothing, sb, m) &\rightarrow_m (is, \vartheta, sb', m) \\
\neg \text{cond (} \vartheta(t \mapsto m \ a)\text{)} &\quad \vartheta' = \vartheta(t \mapsto m \ a) \\
(\text{RMW a t (D, f) cond ret A L R W # is, } \varnothing, [], m) &\rightarrow_m (is, \vartheta', [], m) \\
\text{cond (} \vartheta(t \mapsto m \ a)\text{)} &\quad \vartheta' = \vartheta(t \mapsto \text{ret (m a)} (f (\vartheta(t \mapsto m \ a)))) \\
(\text{RMW a t (D, f) cond ret A L R W # is, } \varnothing, [], m) &\rightarrow_m (is, \vartheta', [], m') \\
(\text{Fence } # \text{ is, } \varnothing, [], m) &\rightarrow_m (is, \vartheta, [], m) \\
(\text{Ghost A L R W # is, } \varnothing, sb, m) &\rightarrow_m (is, \vartheta, sb, m)
\end{align*}\]

Fig. 4: Memory transitions of store buffer machine

The memory transition are defined in Figure 4. With \text{buffered-val sb a} we obtain the value of the last write to address a which is still pending in the store buffer. In case no outstanding write is in the store buffer we read from the memory. Store operations have no immediate effect on the memory but are queued in the store buffer instead. Interlocked operations and the fence operation require an empty store buffer, which means that it has to be flushed before the action can take place. The read-modify-write instruction first adds the current value at address a to temporary t and then checks the store condition \text{cond} on the temporaries. If it fails this read is the final result of the operation. Otherwise the store is performed. The resulting value of the temporary t is specified by the function \text{ret} which considers both the old and new value as input. The fence and the ghost instruction are just skipped.

4.2 Virtual machine

The virtual machine is a sequentially consistent machine without store buffers, maintaining additional ghost state to check for the programming discipline. A thread configuration is a tuple \((p, is, \varnothing, D, O)\), with a dirty flag \(D\) indicating whether there may be an outstanding volatile write in the store buffer and the set of owned addresses \(O\). The dirty flag \(D\) is considered to specify if a read is clean: for all volatile reads the dirty flag must not be set. The global configuration of the virtual machine \((ts, m, S)\) maintains a Boolean map of shared addresses \(S\) (indicating write permission). Addresses in the domain of mapping \(S\) are considered shared and the set of read-only addresses is obtained from \(S\) by: \text{read-only } S \equiv \{a. S a = [\text{False}]\}

According to the rules in Fig 5 a global transition of the virtual machine \((ts, m, S) \Rightarrow (ts', m', S')\) is either a program or a memory step. The transition rules for its memory system are defined in Figure 6. In addition to the transition rules for the virtual machine we introduce the \text{safety} judgment \(Os,i \vdash (is, \varnothing, m, D, O, S) \checkmark\) in Figure 7, where \(Os\) is the list of ownership sets obtained from the thread list \(ts\) and \(i\) is the index of the current
\[
\begin{align*}
i < |ts| & \quad ts|_i = (p, \text{ is }, \emptyset, \mathcal{D}, \mathcal{O}) \quad \emptyset \vdash p \rightarrow_t (p', \text{ is}') \\
(t_s, m, S) & \Rightarrow (ts[i := (p', \text{ is }' \oplus \emptyset', \emptyset, \mathcal{D}, \mathcal{O})], m, S)
\end{align*}
\]

Fig. 5: Global transitions of virtual machine

\[
\begin{align*}
(\text{Read volatile a t} \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset(t \mapsto m a), x, m, \text{ghost}) \\
(\text{Write False a} (D, f) A L R W \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset, x, m(a := f \emptyset), \text{ghost}) \\
\text{ghost} & = (D, \mathcal{O}, S) \\
\text{ghost}' & = (\text{True, } \mathcal{O} \cup A - R, S \oplus W R \ominus A L) \\
(\text{Write True a} (D, f) A L R W \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset, x, m(a := f \emptyset), \text{ghost}') \\
\neg\text{ cond } (\emptyset(t \mapsto m a)) & \quad \text{ghost} = (D, \mathcal{O}, S) \\
\text{ghost}' & = (\text{False, } \mathcal{O}, S) \\
(\text{RMW a t} (D, f) \text{ cond ret} A L R W \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset', x, m', \text{ghost}') \\
\text{cond } (\emptyset(t \mapsto m a)) & \quad \emptyset' = \emptyset(t \mapsto \text{ret } (m a)) \\
m' & = m(a := f (\emptyset(t \mapsto m a))) \\
\text{ghost} & = (D, \mathcal{O}, S) \\
\text{ghost}' & = (\text{False, } \mathcal{O} \cup A - R, S \oplus W R \ominus A L) \\
(\text{RMW a t} (D, f) \text{ cond ret} A L R W \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset', x, m', \text{ghost}') \\
\text{ghost} & = (D, \mathcal{O}, S) \\
\text{ghost}' & = (\text{False, } \mathcal{O}, S) \\
(\text{Fence } \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset, x, m, \text{ghost'}) \\
\text{ghost} & = (D, \mathcal{O}, S) \\
\text{ghost}' & = (\text{False, } \mathcal{O} \cup A - R, S \oplus W R \ominus A L) \\
(\text{Ghost} A L R W \# \text{ is}, \emptyset, x, m, \text{ghost}) & \Rightarrow_m (\text{is}, \emptyset, x, m, \text{ghost'})
\end{align*}
\]

Fig. 6: Memory transitions of the virtual machine
thread. Safety of all reachable states of the virtual machine ensures that the programming discipline is obeyed by the program and is our formal prerequisite for the simulation theorem. It is left as a proof obligation to be discharged by means of a proper program logic for sequentially consistent executions. In the following we elaborate on the rules of

\[
\begin{align*}
& a \in \mathcal{O} \lor a \in \text{read-only } S \lor \text{volatile } \land a \in \text{dom } S \quad \text{volatile } \rightarrow \neg D \\
& \quad \text{Os},i \vdash (\text{Read volatile } a \; \#\; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad a \in \mathcal{O} \quad a \notin \text{dom } S \\
& \quad \text{Os},i \vdash (\text{Write False } a \; (D, \; f) \; A \; L \; R \; W \; \# \; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad \forall j < |\text{Os}|. \; i \neq j \rightarrow a \notin \text{Os}_{ij} \quad a \notin \text{read-only } S \\
& \quad \forall j < |\text{Os}|. \; i \neq j \rightarrow A \cap \text{Os}_{ij} = \emptyset \quad A \subseteq \mathcal{O} \cup \text{dom } S \quad L \subseteq A \quad R \subseteq \mathcal{O} \quad A \cap R = \emptyset \\
& \quad \text{Os},i \vdash (\text{Write True } a \; (D, \; f) \; A \; L \; R \; W \; \# \; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad \neg \text{cond} \left( \emptyset(t \rightarrow m \; a) \right) \quad a \in \text{dom } S \cup \mathcal{O} \\
& \quad \text{Os},i \vdash (\text{RMW } a \; (D, \; f) \; \text{cond ret } A \; L \; R \; W \; \# \; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad \text{cond} \left( \emptyset(t \rightarrow m \; a) \right) \quad \forall j < |\text{Os}|. \; i \neq j \rightarrow a \notin \text{Os}_{ij} \quad a \notin \text{read-only } S \\
& \quad \forall j < |\text{Os}|. \; i \neq j \rightarrow A \cap \text{Os}_{ij} = \emptyset \quad A \subseteq \mathcal{O} \cup \text{dom } S \quad L \subseteq A \quad R \subseteq \mathcal{O} \quad A \cap R = \emptyset \\
& \quad \text{Os},i \vdash (\text{RMW } a \; (D, \; f) \; \text{cond ret } A \; L \; R \; W \; \# \; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad \text{cond} \left( \emptyset(t \rightarrow m \; a) \right) \quad \forall j < |\text{Os}|. \; i \neq j \rightarrow A \cap \text{Os}_{ij} = \emptyset \\
& \quad \text{Os},i \vdash (\text{Fence } \# \; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad A \subseteq \text{dom } S \cup \mathcal{O} \quad L \subseteq A \quad R \subseteq \mathcal{O} \quad A \cap R = \emptyset \\
& \quad \text{Os},i \vdash (\text{Ghost } A \; L \; R \; W \; \# \; is, \; \emptyset, \; m, \; D, \; \mathcal{O}, \; \mathcal{S}) \lor \\
& \quad \forall j < |\text{Os}|. \; i \neq j \rightarrow A \cap \text{Os}_{ij} = \emptyset \\
\end{align*}
\]

Fig. 7: Safe configurations of a virtual machine

Figures 6 and 7 in parallel. To read from an address it either has to be owned or read-only or it has to be volatile and shared. Moreover the read has to be clean. The memory content of address \(a\) is stored in temporary \(t\). Non-volatile writes are only allowed to owned and unshared addresses. The result is written directly into the memory. A volatile write is only allowed when no other thread owns the address and the address is not marked as read-only. Simultaneously with the volatile write we can transfer ownership as specified by the annotations \(A, \; L, \; R\) and \(W\). The acquired addresses \(A\) must not be owned by any other thread and stem from the shared addresses or are already owned. Reacquiring owned addresses can be used to change the shared-status via the set of local addresses \(L\) which have to be a subset of \(A\). The released addresses \(R\) have to be owned and distinct from the acquired addresses \(A\). After the write the new ownership set of the thread is obtained by adding the acquired addresses \(A\) and releasing the addressing \(R, \; O \cup A \rightarrow R\).

The released addresses \(R\) are augmented to the shared addresses \(S\) and the local addresses \(L\) are removed. We also take care about the write permissions in the shared state: the released addresses in set \(W\) as well as the acquired addresses are marked writable: \(S \oplus W \quad R \oplus_A L\). The auxiliary ternary operators to augment and subtract addresses from the sharing map are defined as follows:

\[
S \oplus_W R \equiv \lambda a. \; \text{if } a \in R \text{ then } [a \in W] \text{ else } S \; a \\
S \oplus_A L \equiv \lambda a. \; \text{if } a \in L \text{ then } \bot \text{ else case } S \; a \text{ of } \bot \Rightarrow \bot \mid \text{[writeable]} \Rightarrow [a \in A \lor \text{writeable}] \\
\]

The read-modify-write instruction first adds the current value at address \(a\) to temporary \(t\) and then checks the store condition \text{cond} on the temporaries. If it fails this read is
the final result of the operation. Otherwise the store is performed. The resulting value of
the temporary \( t \) is specified by the function \( \text{ret} \) which considers both the old and new value
as input. As the read-modify-write instruction is an interlocked operation which flushes
the store buffer as a side effect the dirty flag \( D \) is reset. The other effects on the ghost
state and the safety sideconditions are the same as for the volatile read and volatile write,
respectively.

The only effect of the fence instruction in the system without store buffer is to reset
the dirty flag.

The ghost instruction \textit{Ghost A L R W} allows to transfer ownership when no write is
involved i.e., when merely reading from memory. It has the same safety requirements as
the corresponding parts in the write instructions.

4.3 Reduction

The reduction theorem we aim at reduces a computation of a machine with store buffers
to a sequential consistent computation of the virtual machine. We formulate this as a
simulation theorem which states that a computation of the store buffer machine \((ts_{sb}, m) \xrightarrow{\cdot} (ts_{sb}', m')\) can be simulated by a computation of the virtual machine \((ts, m, S) \xrightarrow{\cdot} (ts', m', S')\). The main theorem only considers computations that start in an initial
configuration where all store buffers are empty and end in a configuration where all store
buffers are empty again. A configuration of the store buffer machine is obtained from a
virtual configuration by removing all ghost components and assuming empty store buffers.
This coupling relation between the thread configurations is written as \( ts_{sb} \sim ts \). Moreover,
the precondition \textit{initial} \( ts S \) ensures that the ghost state of the initial configuration of
the virtual machine is properly initialized: the ownership sets of the threads are distinct,
an address marked as read-only (according to \( S \)) is unowned and every unowned address
is shared. Finally with \textit{safe-reach} \((ts, m, S)\) we ensure conformance to the programming
discipline by assuming that all reachable configuration in the virtual machine are safe
(according to the rules in Figure 7).

\textbf{Theorem 1 (Reduction).}

\[ \begin{align*}
(ts_{sb}, m) &\xrightarrow{\cdot} (ts_{sb}', m') \land \text{empty-store-buffers} \land ts_{sb}' \sim ts \land \text{initial, } ts S \land \\
\text{safe-reach} \ &((ts, m, S) \rightarrow \\
\exists ts' S'. \ &((ts, m, S) \xrightarrow{\cdot} (ts', m', S') \land ts_{sb}' \sim ts')
\end{align*} \]

This theorem captures our intuition that every result that can be obtained from a com-
putation of the store buffer machine can also be obtained by a sequentially consistent
computation. However, to prove it we need some generalizations that we sketch in the
following sections. First of all the theorem is not inductive as we do not consider arbitrary
intermediate configurations but only those where all store buffers are empty. For interme-
diate configurations the coupling relation becomes more involved. The major obstacle is
that a volatile read (from memory) can overtake non-volatile writes that are still in the
store-buffer and have not yet emerged to memory. Keep in mind that our programming
discipline only ensures that no \textit{volatile} writes can be in the store buffer the moment we do
a volatile read, outstanding non-volatile writes are allowed. This reordering of operations
is reflected in the coupling relation for intermediate configurations as discussed in the
following section.

5 Building blocks of the proof

A corner stone of the proof is a proper coupling relation between an \textit{intermediate} configuration
of a machine with store buffers and the virtual machine without store buffers. It
allows us to simulate every computation step of the store buffer machine by a sequence
do of steps (potentially empty) on the virtual machine. This transformation is essentially a
sequentialization of the trace of the store buffer machine. When a thread of the store
buffer machine executes a non-volatile operation, it only accesses memory which is not
modified by any other thread (it is either owned or read-only). Although a non-volatile
store is buffered, we can immediately execute it on the virtual machine, as there is no
competing store of another thread. However, with volatile writes we have to be careful,
since concurrent threads may also compete with some volatile write to the same address.
At the moment the volatile write enters the store buffer we do not yet know when it will
be issued to memory and how it is ordered relatively to other outstanding writes of other
threads. We therefore have to suspend the write on the virtual machine from the moment
it enters the store buffer to the moment it is issued to memory. For volatile reads our
programming discipline guarantees that there is no volatile write in the store buffer by
flushing the store buffer if necessary. So there are at most some outstanding non-volatile
writes in the store buffer, which are already executed on the virtual machine, as described
before. One simple coupling relation one may think of is to suspend the whole store buffer
as not yet executed instructions of the virtual machine. However, consider the following
scenario. A thread is reading from a volatile address. It can still have non-volatile writes
in its store buffer. Hence the read would be suspended in the virtual machine, and other
writes to the address (e.g. interlocked or volatile writes of another thread) could invalidate
the value. Altogether this suggests the following refined coupling relation: the state of the
virtual machine is obtained from the state of the store buffer machine, by executing each
store buffer until we reach the first volatile write. The remaining store buffer entries are
suspended as instructions. As we only execute non volatile writes the order in which we
execute the store buffers should be irrelevant. This coupling relation allows a volatile read
to be simulated immediately on the virtual machine as it happens on the store buffer
machine.

From the viewpoint of the memory the virtual machine is ahead of the store buffer
machine, as leading non-volatile writes already took effect on the memory of the virtual
machine while they are still pending in the store buffer. However, if there is a volatile write
in the store buffer the corresponding thread in the virtual machine is suspended until the
write leaves the store buffer. So from the viewpoint of the already executed instructions
the store buffer machine is ahead of the virtual machine. To keep track of this delay we
introduce a variant of the store buffer machine below, which maintains the history of
executed instructions in the store buffer (including reads and program steps). Moreover,
the intermediate machine also maintains the ghost state of the virtual machine to support
the coupling relation. We also introduce a refined version of the virtual machine below,
which we try to motivate now. Essentially the programming discipline only allows races
between volatile (or interlocked) operations. By race we mean two competing memory
accesses of different threads of which at least one is a write. For example the discipline
guarantees that a volatile read may not be invalidated by a non-volatile write of another
thread. While proving the simulation theorem this manifests in the argument that a read
of the store-buffer machine and the virtual machine sees the same value in both machines:
the value seen by a read in the store buffer machine stays valid as long as it has not yet
made its way out in the virtual machine. To rule out certain races from the execution
traces we make use of the programming discipline, which is formalized in the safety of all
reachable configurations of the virtual machine. Some races can be ruled out by continuing
the computation of the virtual machine until we reach a safety violation. However, some
cannot be ruled out by the future computation of the current trace, but can be invalidated
by a safety violation of another trace that deviated from the current one at some point
in the past. Consider two threads. Thread 1 attempts to do a volatile read from address a which is currently owned (and not shared) by thread 2, which attempts to do a non-volatile write on a with value 42 and then release the address. In this configuration there is already a safety violation. Thread 1 is not allowed to perform a volatile read from an address that is not shared. However, when Thread 2 has executed his update and has released ownership (both are non-volatile operations) there is no safety violation anymore. Unfortunately this is the state of the virtual machine when we consider the instructions of Thread 2 to be in the store buffer. The store buffer machine and the virtual machine are out of sync. Whereas in the virtual machine Thread 1 will already read 42 (all non-volatile writes are already executed in the virtual machine), the non-volatile write may still be pending in the store buffer of Thread 2 and hence Thread 1 reads the old value in the store buffer machine. This kind of issues arise when a thread has released ownership in the middle of non-volatile operations of the virtual machine, but the next volatile write of this thread has not yet made its way out of the store buffer. When another thread races for the released address in this situation there is always another scheduling of the virtual machine where the release has not yet taken place and we get a safety violation. To make these safety violations visible until the next volatile write we introduce another ghost component that keeps track of the released addresses. It is augmented when an ghost operation releases an address and is reset as the next volatile write is reached. Moreover, we refine our rules for safety to take these released addresses into account. For example, a write to an released address of another thread is forbidden. We refer to these refined model as delayed releases (as no other thread can acquire the address as long as it is still in the set of released addresses) and to our original model as free flowing releases (as the effect of a release immediate takes place at the point of the ghost instruction). Note that this only affects ownership transfer due to the Ghost instruction. Ownership transfer together with volatile (or interlocked) writes happen simultaneously in both models.

Note that the refined rules for delayed releases are just an intermediate step in our proof. They do not have to be considered for the final programming discipline. As sketched above we can show in a separate theorem that a safety violation in a trace with respect to delayed releases implies a safety violation of a (potentially other) trace with respect to free flowing releases. Both notions of safety collaps in all configurations where there are no released addresses, like the initial state. So if all reachable configurations are safe with respect to free flowing releases they are also safe with respect to delayed releases. This allows us to use the stricter policy of delayed releases for the simulation proof. Before continuing with the coupling relation, we introduce the refined intermediate models for delayed releases and store buffers with history information.

5.1 Intermediate models

We begin with the virtual machine with delayed releases, for which the memory transitions \((is, \theta, m, D, O, R, S) \xrightarrow{m} (is', \theta', m', D', O', R', S')\) are defined Figure 8. The additional ghost component \(R\) is a mapping from addresses to a Boolean flag. If an address is in the domain of \(R\) it was released. The boolean flag is considered to figure out if the released address was previously shared or not. In case the flag is True it was shared otherwise not. This subtle distinction is necessary to properly handle volatile reads. A volatile read from an address owned by another thread is fine as long as it is marked as shared. The released addresses \(R\) are reset at every volatile write as well as interlocked operations and the fence instruction. They are augmented at the ghost instruction taking the sharing information into account:

\[ \text{aug} (\text{dom} S) R R = \]
(Read volatile a t ≠ is, ə, m, ghst) \overset{\rightarrow}{\rightarrow}_m (is, ə(t → m a), m, ghst)

(Write False a (D, f) A L R W ≠ is, ə, m, ghst) \overset{\rightarrow}{\rightarrow}_m (is, ə, m(a := f ə), ghst)

ghst = (D, O, R, S)      ghst' = (True, O ∪ A − R, empty, S ⊖_W R ⊖_A L)

(Write True a (D, f) A L R W ≠ is, ə, m, ghst) \overset{\rightarrow}{\rightarrow}_m (is, ə, m(a := f ə), ghst')

~ cond (ə(t → m a))      ghst = (D, O, R, S)      ghst' = (False, O ∩ A − R, empty, S ⊖_W R ⊖_A L)

(RMW a t (D, f) cond ret A L R W ≠ is, ə, m, ghst) \overset{\rightarrow}{\rightarrow}_m (is, ə', m', ghst')

cond (ə(t → m a))   ə' = ə(t → ret (m a) (f (ə(t → m a))))   m' = m(a := f (ə(t → m a)))

ghst = (D, O, R, S)      ghst' = (False, O ∩ A − R, empty, S ⊖_W R ⊖_A L)

(GHOST A L R W ≠ is, ə, m, ghst) \overset{\rightarrow}{\rightarrow}_m (is, ə, m, ghst')

Fig. 8: Memory transitions of the virtual machine with delayed releases

(λa. if a ∈ R then case R a of ⊥ ⇒ |a ∈ dom S| | |s| ⇒ |s ∧ a ∈ dom S|
else R a)

If an address is freshly released (a ∈ R and R a = ⊥) the flag is set according to dom S. Otherwise the flag becomes |False| in case the released address is currently unshared. Note that with this definition R a = |False| stays stable upon every new release and we do not lose information about a release of an unshared address.

The global transition (ts, m, s) \overset{\rightarrow}{\rightarrow}_s (ts', m', s') are analogous to the rules in Figure 5 replacing the memory transitions with the refined version for delayed releases.

The safety judgment for delayed releases OsRsSi→ (is, ə, m, D, O, S) \overset{\rightarrow}{\rightarrow}_s is defined in Figure 9. Note the additional component Rs which is the list of release maps of all threads. The rules are strict extensions of the rules in Figure 7: writing or acquiring an address a is only allowed if the address is not in the release set of another thread (a ∉ dom Rs[is]); reading from an address is only allowed if it is not released by another thread while it was unshared (Rs[a] = |False|).

For the store buffer machine with history information we not only write into the store buffer but also record reads, program steps and ghost operations. This allows us to restore the necessary computation history of the store buffer machine and relate it to the virtual machine which may fall behind the store buffer machine during execution. Altogether an entry in the store buffer is either a

- READsb volatile a t v, recording a corresponding read from address a which loaded the value v to temporary t, or a
- WRITEsb volatile a sop v for an outstanding write, where operation sop evaluated to value v, or of the form
- PROGs sb p p' is', recording a program transition from p to p' which issued instructions is', or of the form
- GHOSTsb A L R W, recording a corresponding ghost operation.

As defined in Figure 10 a write updates the memory when it exits the store buffer, all other store buffer entries may only have an effect on the ghost state. The effect on the ownership
\[ a \in O \lor a \in \text{read-only} S \lor \text{volatile} \land a \in \text{dom} S \quad \forall j < |O_s|, i \neq j \rightarrow R_{s[j]} a \neq \text{[False]} \]

\[ \neg \text{volatile} \rightarrow (\forall j < |O_s|, i \neq j \rightarrow a \notin \text{dom } R_{s[j]}) \quad \text{volatile} \rightarrow \neg D \]

\[ \text{Os, Rs, t} \vdash (\text{Read volatile a t # is}, \emptyset, m, A, O, S) \]

\[ \text{Os, Rs, t} \vdash (\text{Write False a (D, f) A L R W # is}, \emptyset, m, D, O, S) \]

\[ \forall j < |O_s|, i \neq j \rightarrow a \notin \text{Os[j]} \cup \text{dom } R_{s[j]} \]

\[ a \notin \text{read-only } S \quad \forall j < |O_s|, i \neq j \rightarrow A \cap (\text{Os[j]} \cup \text{dom } R_{s[j]}) = \emptyset \]

\[ A \subseteq \text{dom } S \cup O \quad L \subseteq A \quad R \subseteq O \quad A \cap R = \emptyset \]

\[ \text{Os, Rs, t} \vdash (\text{Write True a (D, f) A L R W # is}, \emptyset, m, D, O, S) \]

\[ \neg \text{cond } (\emptyset(t \rightarrow m a)) \quad a \in \text{dom } S \cup O \quad \forall j < |O_s|, i \neq j \rightarrow R_{s[j]} a \neq \text{[False]} \]

\[ \text{Os, Rs, t} \vdash (\text{RMW a t (D, f) cond ret A L R W # is}, \emptyset, m, D, O, S) \]

\[ \text{Os, Rs, t} \vdash (\text{Fence # is}, \emptyset, m, D, O, S) \]

\[ A \subseteq \text{dom } S \cup O \]

\[ L \subseteq A \quad R \subseteq O \quad A \cap R = \emptyset \quad \forall j < |O_s|, i \neq j \rightarrow A \cap (\text{Os[j]} \cup \text{dom } R_{s[j]}) = \emptyset \]

\[ \text{Os, Rs, t} \vdash (\text{Ghost A L R W # is}, \emptyset, m, D, O, S) \]

\[ \text{Os, Rs, t} \vdash (\emptyset, \emptyset, m, D, O, S) \]

Fig. 9: Safe configurations of a virtual machine (delayed-releases)

\[ (m, \text{Write}_{ab} \text{ False a sop v A L R W # sb, O, R, S}) \rightarrow_{abh} (m(a := v), \text{sb, O, R, S}) \]

\[ \text{Os'} = \text{O} \cup A - R \quad \text{S'} = \text{S} \oplus_{\text{W}} R \ominus_{\text{A}} \text{L} \]

\[ (m, \text{Write}_{ab} \text{ True a sop v A L R W # sb, O, R, S}) \rightarrow_{abh} (m(a := v), \text{sb, O', empty, S'}) \]

\[ (m, \text{Read}_{ab} \text{ volatile a t v # sb, O, R, S}) \rightarrow_{abh} (m, \text{sb, O, R, S}) \]

\[ (m, \text{Prog}_{ab} p p' \text{ is # sb, O, R, S}) \rightarrow_{abh} (m, \text{sb, O, R, S}) \]

\[ \text{Os'} = \text{O} \cup A - R \quad \text{R'} = \text{aug (dom S)} R \text{R} \quad \text{S'} = \text{S} \oplus_{\text{W}} R \ominus_{\text{A}} \text{L} \]

Fig. 10: Store buffer transitions with history
information is analogous to the corresponding operations in the virtual machine. The memory transitions defined in Figure 11 are straightforward extensions of the store buffer transitions of Figure 11 augmented with ghost state and recording history information in the store buffer. Note how we deal with the ghost state. Only the dirty flag is updated when the instruction enters the store buffer, the ownership transfer takes effect when the instruction leaves the store buffer. The global transitions \((t_{sbh}, m, S) \xrightarrow{abh} (t_{sbh}', m', S')\)

\[
v = \text{(case buffered-val sb a of } \bot \Rightarrow m a | \{v' \Rightarrow v\})\quad sb' = sb \oplus \text{[Read}_{abh} \text{ volatile a t v]}
\]

\[
\text{(Read volatile a t # is, } \varnothing, \text{ sb, m, ghst) } \xrightarrow{abh}_{m} (\text{is, } \varnothing(t \rightarrow v), \text{ sb}', m, \text{ ghst})
\]

\[
\text{(Write False a (D, f) A L R W # is, } \varnothing, \text{ sb, m, ghst) } \xrightarrow{abh}_{m} (\text{is, } \varnothing, \text{ sb}', m, \text{ ghst})
\]

\[
\text{ghst} = (D, \varnothing, R, S)\quad \text{ghst}' = (\text{True, } O, R, S)
\]

\[
\text{(Write True a (D, f) A L R W # is, } \varnothing, \text{ sb, m, ghst) } \xrightarrow{abh}_{m} (\text{is, } \varnothing, \text{ sb}', m, \text{ ghst'})
\]

\[
\sim \text{ cond (}\varnothing(t \rightarrow m a)\)\quad \text{ghst} = (D, \varnothing, R, S)\quad \text{ghst}' = (\text{False, } O, \text{ empty, } S)
\]

\[
\text{(RMW a t (D, f) cond ret A L R W # is, } \varnothing, \text{ [], m, ghst) } \xrightarrow{abh}_{m} (\text{is, } \varnothing(t \rightarrow m a), \text{ [], m, ghst'})
\]

\[
\text{cond (}\varnothing(t \rightarrow m a)\)\quad \varnothing' = \varnothing(t \rightarrow \text{ret (m a) (f (\varnothing(t \rightarrow m a)))))\quad m' = m(a := f (\varnothing(t \rightarrow m a)))
\]

\[
\text{ghst} = (D, \varnothing, R, S)\quad \text{ghst}' = (\text{False, } O \cup A - R, \text{ empty, } S \oplus_{w} R \ominus_{A} L)
\]

\[
\text{(Fence # is, } \varnothing, \text{ [], m, D, O, R, S) } \xrightarrow{abh}_{m} (\text{is, } \varnothing, \text{ [], m, False, } O, \text{ empty, } S)
\]

\[
\text{(Ghost A L R W # is, } \varnothing, \text{ sb, m, G) } \xrightarrow{abh}_{m} (\text{is, } \varnothing, \text{ sb }\oplus_{\text{[Ghost}_{abh} A L R W], m, G})
\]

Fig. 11: Memory transitions of store buffer machine with history

are analogous to the rules in Figure 2 replacing the memory transitions and store buffer transitions accordingly.

5.2 Coupling relation

After this introduction of the immediate models we can proceed to the details of the coupling relation, which relates configurations of the store buffer machine with history and the virtual machine with delayed releases. Remember the basic idea of the coupling relation: the state of the virtual machine is obtained from the state of the store buffer machine, by executing each store buffer until we reach the first volatile write. The remaining store buffer entries are suspended as instructions. The instructions now also include the history entries for reads, program steps and ghost operations. The suspended reads are not yet visible in the temporaries of the virtual machine. Similar the ownership effects (and program steps) of the suspended operations are not yet visible in the virtual machine. The coupling relation between the store buffer machine and the virtual machine is illustrated in Figure 12. The thread issue instructions to the store buffers from the right and the instructions emerge from the store buffers to main memory from the left. The dotted line illustrates the state of the virtual machines memory. It is obtained from the memory of the store buffer machine by executing the purely non-volatile prefixes of the store buffers. The remaining entries of the store buffer are still (suspended) instructions in the virtual machine.
Consider the following configuration of a thread $ts_{sbh}[j]$ in the store buffer machine, where $i_k$ are the instructions and $s_k$ the store buffer entries. Let $s_v$ be the first volatile write in the store buffer. Keep in mind that new store buffer entries are appended to the end of the list and entries exit the store buffer and are issued to memory from the front of the list.

$$
ts_{sbh}[j] = (p, [i_1, \ldots, i_n], \vartheta, [s_1, \ldots, s_v, s_{v+1}, \ldots, s_m], \mathcal{D}, O, R)
$$

The corresponding configuration $ts[j]$ in the virtual machine is obtained by suspending all store buffer entries beginning at $s_v$ to the front of the instructions. A store buffer READ$_{sb}$ / WRITE$_{sb}$ / GHOST$_{sb}$ is converted to a READ / WRITE / GHOST instruction. We take the freedom to make this coercion implicit in the example. The store buffer entries preceding $s_v$ have already made their way to memory, whereas the suspended read operations are not yet visible in the temporaries $\vartheta'$. Similar, the suspended updates to the ownership sets and dirty flag are not yet recorded in $O', R'$ and $D'$.

$$
ts[j] = (p, [s_v, s_{v+1}, \ldots, s_m, i_1, \ldots, i_n], \vartheta', D', O', R')
$$

This example illustrates that the virtual machine falls behind the store buffer machine in our simulation, as store buffer instructions are suspended and reads (and ghost operations) are delayed and not yet visible in the temporaries (and the ghost state). This delay can also propagate to the level of the programming language, which communicates with the memory system by reading the temporaries and issuing new instructions. For example the control flow can depend on the temporaries, which store the result of branching conditions. It may happen that the store buffer machine already has evaluated the branching condition by referring to the values in the store buffer, whereas the virtual machine still has to wait. Formally this manifests in still undefined temporaries. Now consider that the program in the store buffer machine makes a step from $p$ to $(p', is')$, which results in a thread configuration where the program state has switched to $p'$, the instructions $is'$ are appended and the program step is recorded in the store buffer:

$$
ts_{sbh}'[j] = (p', [i_1, \ldots, i_n] @ is', \vartheta', [s_1, \ldots, s_v, \ldots, s_m, \text{PROG}_sb \ p \ p' \ is'], \mathcal{D}, O, R)
$$

The virtual machine however makes no step, since it still has to evaluate the suspended instructions before making the program step. The instructions $is'$ are not yet issued and the program state is still $p$. We also take these program steps into account in our final coupling relation $(ts_{sbh}, m_{sbh}, S_{sbh}) \sim (ts, m, S)$, defined in Figure 13. We denote the already simulated store buffer entries by execs and the suspended ones by suspends. The function instrs converts them back to instructions, which are a prefix of the instructions of the virtual...
reads from θ is @ but not yet on the virtual machine. This situation is formalized as instructions have already made their way to the instructions of the store buffer machine but still recorded in the remainder of the store buffer with function \( \text{prog-instrs} \). We collect the additional instructions which were issued by program instructions \( \text{exec-all-until-volatile-write} \) local, and hence could be executed in any order with the same result on the memory.

write is hit, excluding it. Thereby only non-volatile writes are executed, which are all thread share-all-until-volatile-write volatile write via the function \( \text{sharing map of the virtual machine is obtained by executing all store buffers until the first \( \text{execs} = \text{takeWhile not-volatile-write sb}; \text{suspends} = \text{dropWhile not-volatile-write sb} \)

\( \text{in } \exists \text{ is } \text{D}. \text{instrs suspends } @ \text{i}_{sbh} = \text{is } @ \text{prog-instrs suspends } \land \text{D}_{sbh} = (\text{D } \lor \text{refs volatile-Write sb } \neq \emptyset) \land \text{ts}_{ij} = (\text{hd-prog } \text{ρ}_{sbh} \text{suspends}, \text{is}, \text{θ}_{sbh} I(\_ \text{read-maps suspends}), \text{D}, \text{acquire execs } \text{O}_{sbh}, \text{release execs } (\text{dom } \text{S}_{sbh}) \text{R}_{sbh}) \)

\( (\text{ts}_{sbh}, \text{m}_{sbh}, \text{S}_{sbh}) \sim (\text{ts}, \text{m}, \text{S}) \)

Fig. 13: Coupling relation

We collect the additional instructions which were issued by program instructions but still recorded in the remainder of the store buffer with function \( \text{prog-instrs} \). These instructions have already made their way to the instructions of the store buffer machine but not yet on the virtual machine. This situation is formalized as \( \text{instrs suspends } @ \text{i}_{sbh} = \text{is } @ \text{prog-instrs suspends} \), where \( \text{is} \) are the instructions of the virtual machine. The program state of the virtual machine is either the same as in the store buffer machine or the first program state recorded in the suspended part of the store buffer. This state is selected by \( \text{hd-prog} \). The temporaries of the virtual machine are obtained by removing the suspended reads from \( \text{θ} \). The memory is obtained by executing all store buffers until the first volatile write\( @ \text{read-maps suspends} \) and \( \text{takeWhile not-volatile-write sb} \). Similarly the sharing map of the virtual machine is obtained by executing all store buffers until the first volatile write via the function \( \text{share-all-until-volatile-write} \). For the local ownership set \( \text{O}_{sbh} \) the auxiliary function \( \text{acquire} \) calculates the outstanding effect of the already simulated parts of the store buffer. Analogously \( \text{release} \) calculates the effect for the released addresses \( \text{R}_{sbh} \).

5.3 Simulation

Theorem 2 is our core inductive simulation theorem. Provided that all reachable states of the virtual machine (with delayed releases) are safe, a step of the store buffer machine (with history) can be simulated by a (potentially empty) sequence of steps on the virtual machine, maintaining the coupling relation and an invariant on the configurations of the store buffer machine.

Theorem 2 (Simulation).

\( (\text{ts}_{sbh}, \text{m}_{sbh}, \text{S}_{sbh}) \equiv (\text{ts}_{sbh}', \text{m}_{sbh}', \text{S}_{sbh}') \land (\text{ts}_{sbh}, \text{m}_{sbh}, \text{S}_{sbh}) \sim (\text{ts}, \text{m}, \text{S}) \land \text{safereach-delayed} \) \( \text{ts}_{sbh} \) \( \text{S}_{sbh} \) \( \text{m}_{sbh} \rightarrow \)

\( \text{invariant } \text{ts}_{sbh}' \text{S}_{sbh}' \text{m}_{sbh}' \land \)

\( (\exists \text{ts}' \text{S}' \text{m}' (\text{ts}, \text{m}, \text{S}) \rightarrow (\text{ts}' \text{m}', \text{S}') \land (\text{ts}_{sbh}', \text{m}_{sbh}', \text{S}_{sbh}') \sim (\text{ts}', \text{m}', \text{S}') ) \)

In the following we discuss the invariant \( \text{ts}_{sbh} \text{S}_{sbh} \text{m}_{sbh} \), where we commonly refer to a thread configuration \( \text{ts}_{sbh[i]} = (p, \text{is}, \text{θ}, \text{sb}, \text{D}, \text{O}, \text{R}) \) for \( i < |\text{ts}_{sbh}| \). By outstanding references we refer to read and write operations in the store buffer. The invariant is a conjunction of several sub-invariants grouped by their content:

\( \text{invariant } \text{ts}_{sbh} \text{S}_{sbh} \text{m}_{sbh} \equiv \text{ownership-inv } \text{S}_{sbh} \text{ts}_{sbh} \land \text{sharing-inv } \text{S}_{sbh} \text{ts}_{sbh} \land \)

\( \forall i < |\text{ts}_{sbh}|. \)
Ownership. (i) For every thread all outstanding non-volatile references have to be owned or refer to read-only memory. (ii) Every outstanding volatile write is not owned by any other thread. (iii) Outstanding accesses to read-only memory are not owned. (iv) The ownership sets of every two different threads are distinct.

Sharing. (i) All outstanding non-volatile writes are unshared. (ii) All unowned addresses are shared. (iii) No thread owns read-only memory. (iv) The ownership annotations of outstanding ghost and write operations are consistent (e.g., released addresses are owned at the point of release). (v) There is no outstanding write to read-only memory.

Temporaries. Temporaries are modeled as an unlimited store for temporary registers. We require certain distinctness and freshness properties for each thread. (i) The temporaries referred to by read instructions are distinct. (ii) The temporaries referred to by reads in the store buffer are distinct. (iii) Read and write temporaries are distinct. (iv) Read temporaries are fresh, i.e., are not in the domain of \( \vartheta \).

Data dependency. Data dependency means that store operations may only depend on previous read operations. For every thread we have: (i) Every operation \((D, f)\) in a write instruction or a store buffer write is valid according to \text{valid-sop}(D, f)\), i.e., function \( f \) only depends on domain \( D \). (ii) For every suffix of the instructions of the form WRITE volatile \( a(D, f)\) \( A \ L R W \ # \ is \) the domain \( D \) is distinct from the temporaries referred to by future read instructions in \( is \). (iii) The outstanding writes in the store buffer do not depend on the read temporaries still in the instruction list.

History. The history information of program steps and read operations we record in the store buffer have to be consistent with the trace. For every thread: (i) The value stored for a non-volatile read is the same as the last write to the same address in the store buffer or the value in memory, in case there is no write in the buffer. (ii) All reads have to be clean. This results from our flushing policy. Note that the value recorded for a volatile read in the initial part of the store buffer (before the first volatile write), may become stale with respect to the memory. Remember that those parts of the store buffer are already executed in the virtual machine and thus cause no trouble. (iii) For every read the recorded value coincides with the corresponding value in the temporaries. (iv) For every WRITE\(ab\) volatile \( a(D, f)\) \( v\ A L R W \ # \ is\) the recorded value \( v\) coincides with \( f\ \vartheta\), and domain \( D \) is subset of \( \text{dom} \vartheta\) and is distinct from the following read temporaries. Note that the consistency of the ownership annotations is already covered by the aforementioned invariants. (v) For every suffix in the store buffer of the form PROG\(ab\) \( p_1 \ p_2 \ is' \ # \ sb'\), either \( p_1 = p \) in case there is no preceding program node in the buffer or it corresponds to the last program state recorded there. Moreover, the program transition \( \vartheta|(-\text{read-trms}\ sb')^+ p_1 \rightarrow_p (p_2, is')\) is possible, i.e., it was possible to execute the program transition at that point. (vi) The program configuration \( p \) coincides with the last program configuration recorded in the store buffer. (vii) As the instructions from a program step are at the one hand appended to the instruction list and on the other hand recorded in the store buffer, we have for every suffix \( sb'\) of the store buffer: \( \exists is'.\ instrs\ sb' @ is = is' @ prog-instrs\ sb'\), i.e., the remaining instructions \( is\) correspond to a suffix of the recorded instructions \( prog-instrs\ sb'\).

Flushes. If the dirty flag is unset there are no outstanding volatile writes in the store buffer.
**Program step.** The program-transitions are still a parameter of our model. In order to make the proof work, we have to assume some of the invariants also for the program steps. We allow the program-transitions to employ further invariants on the configurations, these are modeled by the parameter \textit{valid}. For example, in the instantiation later on the program keeps a counter for the temporaries, for each thread. We maintain distinctness of temporaries by restricting all temporaries occurring in the memory system to be below that counter, which is expressed by instantiating \textit{valid}. Program steps, memory steps and store buffer steps have to maintain \textit{valid}. Furthermore we assume the following properties of a program step: (i) The program step generates fresh, distinct read temporaries, that are neither in \( \emptyset \) nor in the store buffer temporaries of the memory system. (ii) The generated memory instructions respect data dependencies, and are valid according to \textit{valid-sop}.

**Proof sketch.** We do not go into details but rather first sketch the main arguments for simulation of a step in the store buffer machine by a potentially empty sequence of steps in the virtual machine, maintaining the coupling relation. Second we exemplarily focus on some cases to illustrate common arguments in the proof. The first case distinction in the proof is on the global transitions in Figure 2. (i) **Program step:** we make a case distinction whether there is an outstanding volatile write in the store buffer or not. If not the configuration of the virtual machine corresponds to the executed store buffer and we can make the same step. Otherwise the virtual machine makes no step as we have to wait until all volatile writes have exited the store buffer. (ii) **Memory step:** we do case distinction on the rules in Figure 11. For read, non volatile write and ghost instructions we do the same case distinction as for the program step. If there is no outstanding volatile write in the store buffer we can make the step, otherwise we have to wait. When a volatile write enters the store buffer it is suspended until it exits the store buffer. Hence we do no step in the virtual machine. The read-modify-write and the fence instruction can all be simulated immediately since the store buffer has to be empty. (iii) **Store Buffer step:** we do case distinction on the rules in Figure 10. When a read, a non volatile write, a ghost operation or a program history node exits the store buffer, the virtual machine does not have to do any step since these steps are already visible. When a volatile write exits the store buffer, we execute all the suspended operations (including reads, ghost operations and program steps) until the next suspended volatile write is hit. This is possible since all writes are non volatile and thus memory modifications are thread local.

In the following we exemplarily describe some cases in more detail to give an impression on the typical arguments in the proof. We start with a configuration \( c_{sbh} = (t_{sbh}, m_{sbh}, S_{sbh}) \) of the store buffer machine, where the next instruction to be executed is a read of thread \( i \): \textit{READ}_{sb} volatile \ a \ t. The configuration of the virtual machine is \( cfg = (ts, m, S) \). We have to simulate this step on the virtual machine and can make use of the coupling relations \( (t_{sbh}, m_{sbh}, S_{sbh}) \sim (ts, m, S) \), the invariants \textit{invariant} \( t_{sbh} S_{sbh} m_{sbh} \) and the safety of all reachable states of the virtual machine: \textit{safe-reach-delayed} \( (ts, m, S) \). The state of the store buffer machine and the coupling with the volatile machine is depicted in Figure 14. Note that if there are some suspended instructions in thread \( i \), we cannot directly exploit the ‘safety of the read’, as the virtual machine has not yet reached the state where thread \( i \) is poised to do the read. But fortunately we have safety of the virtual machin of all reachable states. Hence we can just execute all suspended instructions of thread \( i \) until we reach the read. We refer to this configuration of the virtual machine as \( cfg' = (ts'', m'', S'') \), which is depicted in Figure 15.

For now we want to consider the case where the read goes to memory and is not forwarded from the store buffer. The value read is \( v = m_{sbh} a \). Moreover, we make a case distinction whether there is an outstanding volatile write in the store buffer of thread \( i \) or
Fig. 14: Thread $i$ poised to read

Fig. 15: Forwarded computation of virtual machine
not. This determines if there are suspended instructions in the virtual machine or not. We
start with the case where there is no such write. This means that there are no suspended
instructions in thread $i$ and therefore $\text{cfg}'' = \text{cfg}$. We have to show that the virtual machine
reads the same value from memory: $v = \text{m} \ a$. So what can go wrong? When can the the
memory of the virtual machine hold a different value? The memory of the virtual machine
is obtained from the memory of the store buffer machine by executing all store buffers
until we hit the first volatile write. So if there is a discrepancy in the value this has to
come from a non-volatile write in the executed parts of another thread, let us say thread
$j$. This write is marked as $x$ in Figure 16.

We refer to $x$ both for the write operation itself and to characterize the point in time
in the computation of the virtual machine where the write was executed. At the point $x$
the write was safe according to rules in Figure 9 for non-volatile writes. So it was owned
by thread $j$ and unshared. This knowledge about the safety of write $x$ is preserved in the
invariants, namely (Ownership.i) and (Sharing.i). Additionally from invariant (Sharing.v)
we know that address $a$ was not read-only at point $x$. Now we combine this information
with the safety of the read of thread $i$ in the current configuration $\text{cfg}$: address $a$ either
has to be owned by thread $i$, or has to be read-only or the read is volatile and $a$ is shared.
Additionally there are the constraints on the released addresses which we will exploit
below. Let us address all cases step by step. First, we consider that address $a$ is currently
owned by thread $i$. As it was owned by thread $j$ at time $x$ there has to be an release of
address $a$ in the executed prefix of the store buffer of thread $j$. This release is recorded in the
release set, so we know $a \in \text{dom} \ R_{s[j]}$. This contradicts the safety of the read. Second, we
consider that address $a$ is currently read-only. At time $x$ address $a$ was owned by thread
$j$, unshared and not read-only. Hence there was a release of address $a$ in the executed
prefix of the store buffer of $j$, where it made a transition unshared and owned to shared.
With the monotonicity of the release sets this means $a \in \text{dom} \ R_{s[j]}$, even more precisely
$R_{s[j]} \ a = [\text{False}]$. Hence there is no chance to get the read safe (neiter a volatile nor a
non-volatile). Third, consider a volatile read and that address $a$ is currently shared. This
is ruled out by the same line of reasoning as in the previous case. So ultimately we have
ruled out all races that could destroy the value at address $a$ and have shown that we can
simulate the step on the virtual machine. This completes the simulation of the case where
there is no store buffer forwarding and no volatile write in the store buffer of thread $i$.
The other cases are handled similar. The main arguments are obtained by arguing about
safety of configuration $\text{cfg}''$ and exploiting the invariants to rule out conflicting operations
in other store buffers. When there is a volatile write in the store buffer of thread \(i\) there are still pending suspended instructions in the virtual machine. Hence the virtual machine makes no step and we have to argue that the simulation relation as well as all invariants still hold.

Up to now we have focused on how to simulate the read and in particular on how to argue that the value read in the store buffer machine is the same as the value read in the virtual machine. Besides these simulation properties another major part of the proof is to show that all invariants are maintained. For example if the non-volatile read enters the store buffer we have to argue that this new entry is either owned or refers to a read-only address (Ownership.i). As for the simulation above this follows from safety of the virtual machine in configuration \(cfg''\). However, consider an ghost operation that acquires an address \(a\). From safety of the configuration \(cfg''\) we can only infer that there is no conflicting acquire in the non-volatile prefixes of the other store buffers. In case an conflicting acquire is in the suspended part of a store buffer of thread \(j\) safety of configuration \(cfg''\) is not enough. But as we have safety of all reachable states we can forward the computation of thread \(j\) until the conflicting acquire is about to be executed and construct an unsafe state which rules out the conflict.

Last we want to comment on the case where the store buffer takes a step. The major case distinction is whether a volatile write leaves the store buffer or not. In the former case the virtual machine has to simulate a whole bunch of instructions at once to simulate the store buffer machine up to the next volatile write in the store buffer. In the latter case the virtual machine does no step at all, since the instruction leaving the store buffer is already simulated. In both cases one key argument is commutativity of non-volatile operations with respect to global effects on the memory or the sharing map. Consider a non-volatile store buffer step of thread \(i\). In the configuration of the virtual machine before the store buffer step of thread \(i\), the simulation relation applies the update to the memory and the sharing map of the store buffer machine, within the operations \(exec\text{-all-until-volatile-write}\) and \(share\text{-all-until-volatile-write}\) ‘somewhere in the middle’ to obtain the memory and the sharing map of the virtual machine. After the store buffer step however, when the non-volatile operations have left the store buffer, the effect is applied to the memory and the sharing map right in the beginning. The invariants and safety sideconditions for non-volatile operations guarantee ‘locality’ of the operation which manifests in commutativity properties. For example, a non-volatile write is thread local. There is no conflicting write in any other store buffer and hence the write can be safely moved to the beginning.

This concludes the discussion on the proof of Theorem 2.

The simulation theorem for a single step is inductive and can therefore be extended to arbitrary long computations. Moreover, the coupling relation as well as the invariants become trivial for a initial configuration where all store buffers are empty and the ghost state is setup appropriately. To arrive at our final Theorem 1 we need the following steps:

1. simulate the computation of the store buffer machine \((ts_{sb}, m) \Rightarrow^* (ts_{sb}'', m'')\) by a computation of a store buffer machine with history \((ts_{sbh}, m, S) \Rightarrow^{sb*} (ts_{sbh}'', m'', S'')\),
2. simulate the computation of the store buffer machine with history by a computation of the virtual machine with delayed releases \((ts, m, S) \models^* (ts', m', S')\) by Theorem 2 (extended to the reflexive transitive closure),
3. simulate the computation of the virtual machine with delayed releases by a computation of the virtual machine with free flowing releases \((ts, m, S) \Rightarrow^* (ts', m', S')\).

\footnote{Here we are sloppy with \(ts\); strictly we would have to distinguish the thread configurations without the \(R\) component from the ones with the \(R\) component used for delayed releases.}
Step 1 is trivial since the bookkeeping within the additional ghost and history state does not affect the control flow of the transition systems and can be easily removed. Similar the additional $R$ ghost component can be ignored in Step 3. However, to apply Theorem 2 in Step 2 we have to convert from safe-reach $(ts, m, S)$ provided by the preconditions of Theorem 1 to the required safe-reach-delayed $(ts, m, S)$. This argument is more involved and we only give a short sketch here. The other direction is trivial as every single case for delayed releases (cf. Figure 9) immediately implies the corresponding case for free flowing releases (cf. Figure 7).

First keep in mind that the predicates ensure that all reachable configurations starting from $(ts, m, S)$ are safe, according to the rules for free flowing releases or delayed releases respectively. We show the theorem by contraposition and start with a computation which reaches a configuration $c$ that is unsafe according to the rules for delayed releases and want to show that there has to be a (potentially other) computation (starting from the same initial state) that leads to an unsafe configuration $c'$ according to free flowing releases. If $c$ is already unsafe according to free flowing releases we have $c' = c$ and are finished. Otherwise we have to find another unsafe configuration. Via induction on the length of the global computation we can also assume that for all shorter computations both safety notions coincide. A configuration can only be unsafe with respect to delayed releases and safe with respect to free flowing releases if there is a race between two distinct Threads $i$ and $j$ on an address $a$ that is in the release set $R$ of one of the threads, lets say Thread $i$. For example Thread $j$ attempts to write to an address $a$ which is in the release set of Thread $i$. If the release map would be empty there cannot be such an race (it would simultaneously be unsafe with respect to free flowing releases). Now we aim to find a configuration $c'$ that is also reachable from the initial configuration and is unsafe with respect to free flowing releases. Intuitively this is a configuration where Thread $i$ is rewinded to the state just before the release of address $a$ and Thread $j$ is in the same state as in configuration $c$. Before the release of $a$ the address has to be owned by Thread $i$, which is unsafe according to free flowing releases as well as delayed releases. So we can argue that either Thread $j$ can reach the same state although Thread $i$ is rewinded or we even hit an unsafe configuration before. What kind of steps can Thread $i$ perform between between the free flowing release point (point of the ghost instruction) and the delayed release point (point of next volatile write, interlocked operation or fence at which the release map is emptied)? How can these actions affect Thread $j$? Note that the delayed release point is not yet reached as this would empty the release map (which we know not to be empty). Thus Thread $i$ does only perform reads, ghost instructions, program steps or non-volatile writes. All of these instructions of Thread $i$ either have no influence on the computation of Thread $j$ at all (e.g. a read, program step, non-volatile write or irrelevant ghost operation) or may cause a safety violation already in a shorter computation (e.g. acquiring an address that another thread holds). This is fine for our inductive argument. So either we can replay every step of Thread $j$ and reach the final configuration $c'$ which is now also unsafe according to free flowing releases, or we hit a configuration $c''$ in a shorter computation which violates the rules of delayed as well as free flowing releases (using the induction hypothesis).

6 PIMP

PIMP is a parallel version of IMP [11], a canonical WHILE-language.

An expression $e$ is either (i) Const $v$, a constant value, (ii) Mem volatile $a$, a (volatile) memory lookup at address $a$, (iii) Tmp sop, reading from the temporaries with a operation sop which is an intermediate expression occurring in the transition rules for statements,
(iv) **Unop** \( f e \), a unary operation where \( f \) is a unary function on values, and finally
(v) **Binop** \( f e_1 e_2 \), a binary operation where \( f \) is a binary function on values.

A statement \( s \) is either (i) **SKIP**, the empty statement, (ii) **Assign volatile** \( a e A L R W \), a (volatile) assignment of expression \( e \) to address \( a \) \( A L R W \), atomic compare and swap at address \( e \) \( a \) \( A L R W \), compare and swap expression \( e \) \( s_e \) \( A L R W \), (iv) **SEQ** \( s_1 s_2 \), sequential composition, (v) **Cond** \( e s_1 s_2 \), the if-then-else statement, (vi) **While** \( e s \), the loop statement with condition \( e \), (vii) **SGhost** and \( SFence \) as stubs for the corresponding memory instructions.

The key idea of the semantics is the following: expressions are evaluated by issuing instructions to the memory system, then the program waits until the memory system has made all necessary results available in the temporaries, which allows the program to make another step. Figure 17 defines expression evaluation. The function \( used-tmps e \) calculates the number of temporaries that are necessary to evaluate expression \( e \), where every **MEM** expression accounts to one temporary. With **issue-comp** \( t e \) we obtain the instruction list for expression \( e \) starting at temporary \( t \), whereas **eval-comp** \( t e \) constructs the operation as a pair of the domain and a function on the temporaries.

The program transitions are defined in Figure 18. We instantiate the program state by a tuple \((s, t)\) containing the statement \( s \) and the temporary counter \( t \). To assign an expression \( e \) to an address(-expression) \( a \) we first create the memory instructions for evaluation the address \( a \) and transforming the expression to an operation on temporaries. The temporary counter is incremented accordingly. When the value is available in the temporaries we continue by creating the memory instructions for evaluation of expression \( e \) followed by the corresponding store operation. Note that the ownership annotations can depend on the temporaries and thus can take the calculated address into account.

Execution of compare and swap CAS involves evaluation of three expressions, the address \( a \) the compare value \( c_e \) and the swap value \( s_e \). It is finally mapped to the read-modify-write instruction RMW of the memory system. Recall that execution of RMW first stores the memory content at address \( a \) to the specified temporary. The condition compares this value with the result of evaluating \( c_e \) and writes swap value \( s_a \) if successful. In either case the temporary finally returns the old value read.

Sequential composition is straightforward. An if-then-else is computed by first issuing the memory instructions for evaluation of condition \( e \) and transforming the condition to an operation on temporaries. When the result is available the transition to the first or second statement is made, depending on the result of \( isTrue \). Execution of the loop is defined...
\[ \forall \text{sop. } a \neq \text{Tmp}\ \text{sop}\ \ a' = \text{Tmp} (\text{eval-expr } t\ a) \quad t' = t + \text{used-tmps } a \quad \text{is} = \text{issue-expr } t\ a \]
\[ \vartheta \vdash (\text{Assign } \text{volatile} a\ e\ A\ L\ R\ W, t) \rightarrow_{p} ((\text{Assign } \text{volatile} a' e\ A\ L\ R\ W, t'), \text{is}) \]
\[ D \subseteq \text{dom } \vartheta \quad \text{is} = \text{issue-expr } t\ e\ \emptyset [\text{Write } \text{volatile} (a\ \vartheta) (\text{eval-expr } t\ e) (A\ \vartheta) (L\ \vartheta) (R\ \vartheta) (W\ \vartheta)] \]
\[ \vartheta \vdash (\text{Assign } (\text{Tmp } (D, a)) e\ A\ L\ R\ W, t) \rightarrow_{p} ((\text{Skip}, t + \text{used-tmps } e), \text{is}) \]

∀ sop. a ≠ Tmp sop a' = Tmp (eval-expr t a) t' = t + used-tmps a is = issue-expr t a

\[ \vartheta \vdash (\text{Assign } \text{volatile } e\ A\ L\ R\ W, t) \rightarrow_{p} ((\text{Assign } \text{volatile } a' e\ A\ L\ R\ W, t'), \text{is}) \]

∀ sop. a ≠ Tmp sop a' = Tmp (eval-expr t a) t' = t + used-tmps a is = issue-expr t a

\[ \vartheta \vdash (\text{Assign } (\text{Tmp } (D, a)) e\ A\ L\ R\ W, t) \rightarrow_{p} ((\text{CAS } a' c_s A\ L\ R\ W, t'), \text{is}) \]

\[ D_c \subseteq \text{dom } \vartheta \quad \text{eval-expr } t\ s_e = (D, t) \quad t' = t + \text{used-tmps } s_e \quad \text{cond} = (\lambda \vartheta.\ \text{the } (\vartheta\ t')) = e\ \vartheta \]
\[ \text{ret} = (\lambda v_1 v_2.\ v_1) \quad \text{is} = \text{issue-expr } t\ s_e \emptyset [\text{RMW } (a\ \vartheta) t'(D, t) \text{ cond} \text{ ret} (A\ \vartheta) (L\ \vartheta) (R\ \vartheta) (W\ \vartheta)] \]
\[ \vartheta \vdash (\text{CAS } (\text{Tmp } (D, a)) (\text{Tmp } (D_c, c)) s_e A\ L\ R\ W, t) \rightarrow_{p} ((\text{Skip}, t\ c_s), \text{is}) \]

\[ \vartheta \vdash (\text{SEQ } s_1 s_2, t) \rightarrow_{p} ((\text{SEQ } s_1 s_2, t'), \text{is}) \]

\[ \vartheta \vdash (\text{SEQ } \text{Skip } s_2, t) \rightarrow_{p} ((s_2, t), []) \]

∀ sop. e ≠ Tmp sop e' = Tmp (eval-expr t e) t' = t + used-tmps e is = issue-expr t e

\[ \vartheta \vdash (\text{Cond } e\ s_1 s_2, t) \rightarrow_{p} ((\text{Cond } e' s_1 s_2, t'), \text{is}) \]

\[ D \subseteq \text{dom } \vartheta \quad \text{isTrue } (e\ \vartheta) \]
\[ \vartheta \vdash (\text{Cond } (\text{Tmp } (D, e)) s_1 s_2, t) \rightarrow_{p} ((s_1, t), []) \]

\[ D \subseteq \text{dom } \vartheta \quad \neg \text{isTrue } (e\ \vartheta) \]
\[ \vartheta \vdash (\text{Cond } (\text{Tmp } (D, e)) s_1 s_2, t) \rightarrow_{p} ((s_2, t), []) \]

\[ \vartheta \vdash (\text{While } e\ s, t) \rightarrow_{p} ((\text{Cond } e (\text{SEQ } s (\text{While } e\ s)) \text{ Skip}, t), []) \]

\[ \vartheta \vdash (\text{SGhost } A\ L\ R\ W, t) \rightarrow_{p} ((\text{Skip}, t), [\text{Ghost } (A\ \vartheta) (L\ \vartheta) (R\ \vartheta) (W\ \vartheta)]) \]

\[ \vartheta \vdash (\text{SFence}, t) \rightarrow_{p} ((\text{Skip}, t), [\text{Fence}]) \]

Fig. 18: Program transitions
by stepwise unfolding. Ghost and fence statements are just propagated to the memory system.

To instantiate Theorem 2 with PIMP we define the invariant parameter valid, which has to be maintained by all transitions of PIMP, the memory system and the store buffer. Let $\theta$ be the valuation of temporaries in the current configuration, for every thread configuration $ts_{sb}[i] = ((s, t), is, \theta, sb, D, O)$ where $i < |ts_{sb}|$ we require: (i) The domain of all intermediate $\text{Tmp}(D, f)$ expressions in statement $s$ is below counter $t$. (ii) All temporaries in the memory system including the store buffer are below counter $t$. (iii) All temporaries less than counter $t$ are either already defined in the temporaries $\theta$ or are outstanding read temporaries in the memory system.

For the PIMP transitions we prove these invariants by rule induction on the semantics. For the memory system (including the store buffer steps) the invariants are straightforward. The memory system does not alter the program state and does not create new temporaries, only the PIMP transitions create new ones in strictly ascending order.

7 Conclusion

We have presented a practical and flexible programming discipline for concurrent programs that ensures sequential consistency on TSO machines, such as present x64 architectures. Our approach covers a wide variety of concurrency control, covering locking, data races, single writer multiple readers, read only and thread local portions of memory. We minimize the need for store buffer flushes to optimize the usage of the hardware. Our theorem is not coupled to a specific logical framework like separation logic but is based on more fundamental arguments, namely the adherence to the programming discipline which can be discharged within any program logic using the standard sequential consistent memory model, without any of the complications of TSO.

Related work. Disclaimer. This contribution presents the state of our work from 2010 [8]. Finally, 8 years later, we made the AFP submission for Isabelle2018. This related work paragraph does not thoroughly cover publications that came up in the meantime.

A categorization of various weak memory models is presented in [2]. It is compatible with the recent revisions of the Intel manuals [10] and the revised x86 model presented in [15]. The state of the art in formal verification of concurrent programs is still based on a sequentially consistent memory model. To justify this on a weak memory model often a quite drastic approach is chosen, allowing only coarse-grained concurrency usually implemented by locking. Thereby data races are ruled out completely and there are results that data race free programs can be considered as sequentially consistent for example for the Java memory model [3, 18] or the x86 memory model [15]. Ridge [17] considers weak memory and data-races and verifies Peterson’s mutual exclusion algorithm. He ensures sequential consistency by flushing after every write to shared memory. Burckhardt and Musuvathi [6] describe an execution monitor that efficiently checks whether a sequentially consistent TSO execution has a single-step extension that is not sequentially consistent. Like our approach, it avoids having to consider the store buffers as an explicit part of the state. However, their condition requires maintaining in ghost state enough history information to determine causality between events, which means maintaining a vector clock (which is itself unbounded) for each memory address. Moreover, causality (being essentially graph reachability) is already not first-order, and hence unsuitable for many types of program verification. Closely related to our work is the draft of Owens [14] which also investigates on the conditions for sequential consistent reasoning within TSO. The notion of a triangular-race free trace is established to exactly characterize the traces on
a TSO machine that are still sequentially consistent. A triangular race occurs between a read and a write of two different threads to the same address, when the reader still has some outstanding writes in the store buffer. To avoid the triangular race the reader has to flush the store buffer before reading. This is essentially the same condition that our framework enforces, if we limit every address to be unowned and every access to be volatile. We regard this limitation as too strong for practical programs, where non-volatile accesses (without any flushes) to temporarily local portions of memory (e.g. lock protected data) is common practice. This is our core motivation for introducing the ownership based programming discipline. We are aware of two extensions of our work that were published in the meantime. Chen et al. [7] also take effects of the MMU into account and generalize our reduction theorem to handle programs that edit page tables. Oberhauser [13] improves on the flushing policy to also take non-triangular races into account and facilitates an alternative proof approach.

**Limitations.** There is a class of important programs that are not sequentially consistent but nevertheless correct.

First consider a simple spinlock implementation with a volatile lock \( l \), where \( l == 0 \) indicates that the lock is not taken. The following code acquires the lock:

```c
while(!interlocked_test_and_set(l));
<critical section accessing protected objects>,
```

and with the assignment \( l = 0 \) we can release the lock again. Within our framework address \( l \) can be considered *unowned* (and hence shared) and every access to it is *volatile*. We do not have to transfer ownership of the lock \( l \) itself but of the objects it protects. As acquiring the lock is an expensive interlocked operation anyway there are no additional restrictions from our framework. The interesting point is the release of the lock via the volatile write \( l=0 \). This leaves the dirty bit set, and hence our programming discipline requires a flushing instruction before the next volatile read. If \( l \) is the only volatile variable this is fine, since the next operation will be a lock acquire again which is interlocked and thus flushes the store buffer. So there is no need for an additional fence. But in general this is not the case and we would have to insert a fence after the lock release to make the dirty bit clean again and to stay sequentially consistent. However, can we live without the fence? For the correctness of the mutual-exclusion algorithm we can, but we leave the domain of sequential consistent reasoning. The intuitive reason for correctness is that the threads waiting for the lock do no harm while waiting. They only take some action if they see the lock being zero again, this is when the lock release has made its way out of the store buffer.

Another typical example is the following simplified form of barrier synchronization: each processor has a flag that it writes (with ordinary volatile writes without any flushing) and other processors read, and each processor waits for all processors to set their flags before continuing past the barrier. This is not sequentially consistent – each processor might see his own flag set and later see all other flags clear – but it is still correct.

Common for these examples is that there is only a single writer to an address, and the values written are monotonic in a sense that allows the readers to draw the correct conclusion when they observe a certain value. This pattern is named *Publication Idiom* in Owens work [14].

**Future work.** The first direction of future work is to try to deal with the limitations of sequential consistency described above and try to come up with a more general reduction
theorem that can also handle non sequential consistent code portions that follow some monotonicity rules.

Another direction of future work is to take compiler optimization into account. Our volatile accesses correspond roughly to volatile memory accesses within a C program. An optimizing compiler is free to convert any sequence of non-volatile accesses into a (sequentially semantically equivalent) sequence of accesses. As long as execution is sequentially consistent, equivalence of these programs (e.g., with respect to final states of executions that end with volatile operations) follows immediately by reduction. However, some compilers are a little more lenient in their optimizations, and allow operations on certain local variables to move across volatile operations. In the context of C (where pointers to stack variables can be passed by pointer), the notion of “locality” is somewhat tricky, and makes essential use of C forbidding (semantically) address arithmetic across memory objects.

Acknowledgements

We thank Mark Hillebrand for discussions and feedback on this work and extensive comments on this report.

A Appendix

After the explanatory text in the main body of the document we now show the plain theory files.

theory ReduceStoreBuffer
imports Main
begin

A.1 Memory Instructions

type-synonym addr = nat
type-synonym val = nat
type-synonym tmp = nat

type-synonym tmprs = tmp ⇒ val option
type-synonym sop = tmp set × (tmprs ⇒ val) — domain and function

locale valid-sop =
fixes sop :: sop
assumes valid-sop: \( D f \varnothing, \)
\[ [sop=(D,f); D \subseteq \text{dom } \varnothing] \]
\[ \Rightarrow \]
\[ f \varnothing = f (\varnothing \upharpoonright D) \]

type-synonym memory = addr ⇒ val
type-synonym owns = addr set
type-synonym rels = addr ⇒ bool option
type-synonym shared = addr ⇒ bool option
type-synonym acq = addr set
type-synonym rel = addr set

31
type-synonym lcl = addr set

type-synonym wrt = addr set

type-synonym cond = tmps ⇒ bool

type-synonym ret = val ⇒ val ⇒ val

datatype instr = Read bool addr tmp

| Write bool addr sop acq lcl rel wrt
| RMW addr tmp sop cond ret acq lcl rel wrt
| Fence
| Ghost acq lcl rel wrt

data type instrs = instr list

type-synonym ('p,'sb,'dirty,'owns,'rels) thread-config =

'p × instrs × tmps × 'sb × 'dirty × 'owns × 'rels
type-synonym ('p,'sb,'dirty,'owns,'rels,'shared) global-config =

('p,'sb,'dirty,'owns,'rels) thread-config list × memory × 'shared

definition owned t = (let (p,instrs,θ,sb,D,O,R) = t in O)

lemma owned-simp [simp]: owned (p,instrs,θ,sb,D,O,R) = (O)
    by (simp add: owned-def)

definition O-sb t = (let (p,instrs,θ,sb,D,O,R) = t in (O,sb))

lemma O-sb-simp [simp]: O-sb (p,instrs,θ,sb,D,O,R) = (O,sb)
    by (simp add: O-sb-def)

definition released t = (let (p,instrs,θ,sb,D,O,R) = t in R)

lemma released-simp [simp]: released (p,instrs,θ,sb,D,O,R) = (R)
    by (simp add: released-def)

lemma list-update-id': v = xs ! i ⇒ xs[i := v] = xs
    by simp

lemmas converse-rtranclp-induct5 =

converse-rtranclp-induct [where a=(m,sb,O,R,S) and b=(m',sb',O',R',S'),
    split-rule,consumes 1, case-names refl step]

A.2 Abstract Program Semantics

locale memory-system =

fixes

memop-step :: (instrs × tmps × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒

(instrs × tmps × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒ bool

(- →_m - [60,60] 100) and
storebuffer-step:: (memory × 'sb × 'owns × 'rels × 'shared) ⇒ (memory × 'sb × 'owns × 'rels × 'shared) ⇒ bool and

locale program =
  fixes
  program-step :: tmps ⇒ 'p ⇒ 'p × instrs ⇒ bool (-⇒ - ⇒ [60,60,60] 100)
  — A program only accesses the shared memory indirectly, it can read the temporaries and can output a sequence of memory instructions

locale computation = memory-system + program +
  constrains
  — The constrains are only used to name the types 'sb and 'p
  storebuffer-step:: (memory × 'sb × 'owns × 'rels × 'shared) ⇒ (memory × 'sb × 'owns × 'rels × 'shared) ⇒ bool and

  memop-step ::
    (instrs × tmps × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒
    (instrs × tmps × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒ bool
    and
  program-step :: tmps ⇒ 'p ⇒ 'p × instrs ⇒ bool
  fixes
  record :: 'p ⇒ 'p ⇒ instrs ⇒ 'sb ⇒ 'sb
  begin
  inductive concurrent-step ::
    (p, sb, 'dirty, 'owns, 'rels, 'shared) global-config ⇒ (p, sb, 'dirty, 'owns, 'rels, 'shared)
  global-config ⇒ bool
  (-⇒ -⇒ [60,60] 100)
  where
  Program:
  [i < length ts; tsli = (p, is, 0, sb, D, O, R);
    0 + p -⇒ (p', is', j)]
  ⇒ (ts, m, S) ⇒ (ts[i]:= (p', is@is', 0, record p p' is' sb, D, O, R)], m, S)

  | Memop:
  [i < length ts; tsli = (p, is, 0, sb, D, O, R);
    (is, 0, sb, m, D, O, R, S) ⇒m (is', 0, sb', m', D', O', R', S')]
  ⇒ (ts, m, S) ⇒ (ts[i]:= (p, is', 0, sb', D', O', R')], m', S')

  | StoreBuffer:
  [i < length ts; tsli = (p, is, 0, sb, D, O, R);
    (m, sb, O, R, S) -⇒sb (m', sb', O', R', S')]
  ⇒ (ts, m, S) ⇒ (ts[i] := (p, is, 0, sb', D, O', R')], m', S')

  definition final:: (p, sb, 'dirty, 'owns, 'rels, 'shared) global-config ⇒ bool
  where
  final c = (¬ (∃ c'. c ⇒ c'))

33
lemma store-buffer-steps:
assumes sb-step: storebuffer-step\^\** (m,sb,O,R,S) \rightarrow (m',sb',O',R',S')
shows \(i < \text{length } ts \Rightarrow ts!i = (p,is,\theta,sb,D,O,R,S) \Rightarrow \text{concurrent-step}^\** (ts,m,S) \rightarrow (ts[i:= (p,is,\theta,sb',D,O',R'),m',S'])
using sb-step
proof \(\text{induct rule: converse-rtranclp-induct5}\)
  case refl then show ?case by (simp add: list-update-id)
next
  case (step m sb OR m' sb'O'R'S')
  note i-bound = \(\langle i < \text{length } ts \rangle\)
  note ts-i = \(\langle ts!i = (p,is,\theta,sb,D,O,R,S) \rangle\)
  note step = \(\langle (m,sb,O,R,S) \rightarrow sb (m'',D,O'',R'') \rangle\)
  let ?ts' = ts[i := (p,is,\theta,sb'',D,O'',R'')]
  from StoreBuffer \(\text{OF i-bound ts-i step}\)
  have (ts,m,S) \Rightarrow (?ts',m'',S'').
  also from i-bound have i-bound': \(i < \text{length } ?ts'\) by simp
  from i-bound have ts'-i: ?ts'!i = (p,is,\theta,sb'',D,O'',R'')
    by simp
  from step.hyps (3) \(\text{OF i-bound'} ts'-i\) \(\text{i-bound}\)
  have concurrent-step\^\** (?ts',m'',S'') \rightarrow (ts[i:= (p,is,\theta,sb',D,O',R')],m',S')
    by (simp)
  finally show ?case .
qed

lemma step-preserves-length-ts:
assumes step: (ts,m,S) \Rightarrow (ts',m',S')
shows length ts' = length ts
using step
apply (cases)
apply auto
done
end
lemmas concurrent-step-cases = computation.concurrent-step.cases
[cases set, consumes 1, case-names Program Memop StoreBuffer]

definition augment-shared:: shared \Rightarrow addr set \Rightarrow addr set \Rightarrow shared (- \oplus - [61,1000,50])\[61]
where
\(S \oplus_W S \equiv (\lambda a. \text{if } a \in S \text{ then Some } (a \in W) \text{ else } S) \ a\)

definition restrict-shared:: shared \Rightarrow addr set \Rightarrow addr set \Rightarrow shared (- \ominus - [51,1000,50])\[51]
where
\(S \ominus_A L \equiv (\lambda a. \text{if } a \in L \text{ then None else case } S) \ a \text{ of None } \Rightarrow \text{None}\)
definition read-only :: shared ⇒ addr set
where
read-only S ≡ {a. (S a = Some False)}

definition shared-le:: shared ⇒ shared ⇒ bool (infix ⊆ₕ 50)
where
m₁ ⊆ₕ m₂ ≡ m₁ ⊆ m₂ ∧ read-only m₁ ⊆ read-only m₂

lemma shared-leD: m₁ ⊆ₕ m₂ ⟹ m₁ ⊆ m₂ ∧ read-only m₁ ⊆ read-only m₂
  by (simp add: shared-le-def)

lemma shared-le-map-le: m₁ ⊆ₕ m₂ ⟹ m₁ ⊆ m₂
  by (simp add: shared-le-def)

lemma shared-le-read-only-le: m₁ ⊆ₕ m₂ ⟹ read-only m₁ ⊆ read-only m₂
  by (simp add: shared-le-def)

lemma dom-augment [simp]: dom (m ⊕ₜ S) = dom m ∪ S
  by (auto simp add: augment-shared-def)

lemma augment-empty [simp]: S ⊕ₜ {} = S
  by (simp add: augment-shared-def)

lemma inter-neg [simp]: X ∩− L = X − L
  by blast

lemma dom-restrict-shared [simp]: dom (m ⊖ₜ A L) = dom m − L
  by (auto simp add: restrict-shared-def split: option.splits)

lemma restrict-shared-UNIV [simp]: (m ⊖ₜ UNIV) = Map.empty
  by (auto simp add: restrict-shared-def split: if-split-asm option.splits)

lemma restrict-shared-empty [simp]: (Map.empty ⊖ₜ A L) = Map.empty
  apply (rule ext)
  by (auto simp add: restrict-shared-def split: if-split-asm option.splits)

lemma restrict-shared-in [simp]: a ∈ L ⟹ (m ⊖ₜ A L) a = None
  by (auto simp add: restrict-shared-def split: if-split-asm option.splits)

lemma restrict-shared-out [simp]: a ∉ L ⟹ (m ⊖ₜ A L) a = Some writeable
  by (auto simp add: restrict-shared-def split: if-split-asm option.splits)

lemma restrict-shared-out'[simp]:
a ∉ L ⟹ m a = Some writeable ⟹ (m ⊖ₜ A L) a = Some (a ∈ A ∨ writeable)
  by (simp add: restrict-shared-out)
lemma augment-mono-map: \( A \subseteq_m B \implies (A \oplus_x C) \subseteq_m (B \oplus_x C) \)
by (auto simp add: augment-shared-def map-le-def domIff)

lemma augment-mono-map: \( A \subseteq_s B \implies (A \oplus_x C) \subseteq_s (B \oplus_x C) \)
by (auto simp add: augment-shared-def shared-le-def map-le-def read-only-def dom-def split: option.splits if-split-asm)

lemma restrict-mono-map: \( A \subseteq_s B \implies (A \ominus_x C) \subseteq_s (B \ominus_x C) \)
by (auto simp add: restrict-shared-def shared-le-def map-le-def read-only-def dom-def split: option.splits if-split-asm)

lemma augment-mono-aux: \( \text{dom } A \subseteq \text{dom } B \implies \text{dom } (A \oplus_x C) \subseteq \text{dom } (B \oplus_x C) \)
by auto

lemma restrict-mono-aux: \( \text{dom } A \subseteq \text{dom } B \implies \text{dom } (A \ominus_x C) \subseteq \text{dom } (B \ominus_x C) \)
by auto

lemma read-only-mono: \( S \subseteq_m S' \implies a \in \text{read-only } S \implies a \in \text{read-only } S' \)
by (auto simp add: map-le-def domIff read-only-def dest!: bspec)

lemma in-read-only-restrict-conv:
\( a \in \text{read-only } (S \ominus_A L) = (a \in \text{read-only } S \land a \notin L \land a \notin A) \)
by (auto simp add: read-only-def restrict-shared-def split: option.splits if-split-asm)

lemma in-read-only-augment-conv: \( a \in \text{read-only } (S \oplus_W R) = (\text{if } a \in R \text{ then } a \notin W \text{ else } a \in \text{read-only } S) \)
by (auto simp add: read-only-def augment-shared-def)

lemmas in-read-only-convs = in-read-only-restrict-conv in-read-only-augment-conv

lemma read-only-dom: \( \text{read-only } S \subseteq \text{dom } S \)
by (auto simp add: read-only-def dom-def)

lemma read-only-empty [simp]: \( \text{read-only } \text{Map}.empty = \{\} \)
by (auto simp add: read-only-def)

lemma restrict-shared-fuse: \( S \ominus_A L \ominus_B M = (S \ominus (A \cup B) (L \cup M)) \)
apply (rule ext)
apply (auto simp add: restrict-shared-def split: option.splits if-split-asm)
done

lemma restrict-shared-empty-set [simp]: \( S \ominus \{\} = S \)
apply (rule ext)
apply (auto simp add: restrict-shared-def split: option.splits if-split-asm)
done

definition augment-rels:: \( \text{addr set } \Rightarrow \text{addr set } \Rightarrow \text{rels } \Rightarrow \text{rels} \)
where
augment-rels $S \ R \ R = (\lambda a. \text{ if } a \in R$
then (case $R$ a of
  None $\Rightarrow$ Some $(a \in S)$
  | Some $s$ $\Rightarrow$ Some $(s \land (a \in S)))$
else $R$ a)

\textbf{declare} domIff [iff del]

\section{A.3 Memory Transitions}

\textbf{locale} gen-direct-memop-step =
\textbf{fixes} emp :: 'rels and aug :: owns $\Rightarrow$ rel $\Rightarrow$ 'rels $\Rightarrow$ 'rels
\textbf{begin}
\textbf{inductive} gen-direct-memop-step :: (instrs $\times$ tmps $\times$ unit $\times$ memory $\times$ bool $\times$ owns $\times$ 'rels $\times$ shared ) $\Rightarrow$ (instrs $\times$ tmps $\times$ unit $\times$ memory $\times$ bool $\times$ owns $\times$ 'rels $\times$ shared ) $\Rightarrow$ bool
(- $\rightarrow$ - [60, 60] 100)
\textbf{where}
  \textbf{Read:} (Read volatile a t is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$
  (is, $\theta$ (t$\rightarrow$m a), x, m, $D$, $O$, $R$, $S$)
  \textbf{WriteNonVolatile:} (Write False a (D, f) A L R W # is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$
  (is, $\theta$, x, m(a$:=f$ $\theta$), $D$, $O$, $R$, $S$)
  \textbf{WriteVolatile:} (Write True a (D, f) A L R W # is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$
  (is, $\theta$, x, m(a$:=f$ $\theta$), True, $O \cup A - R$, emp, $S \oplus W R \ominus A L$)
  \textbf{Fence:} (Fence $\#$ is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$ (is, $\theta$, x, m, False, $O$, emp, $S$)
  \textbf{RMWReadOnly:} $\neg \text{ cond } (\theta(t\rightarrow m a))$ $\implies$
  (RMW a t (D, f) cond ret A L R W # is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$ (is, $\theta$(t$\rightarrow$m a), x, m,
  False, $O$, emp, $S$)
  \textbf{RMWWrite:} $\text{ cond } (\theta(t\rightarrow m a))$ $\implies$
  (RMW a t (D, f) cond ret A L R W # is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$
  (is, $\theta$(t$\rightarrow$ ret (m a) (f($\theta$(t$\rightarrow$m a)))), x, m(a$:=f$($\theta$(t$\rightarrow$m a)))), False, $O \cup A - R$, emp, $S \oplus_{W R \ominus A L}$
  \textbf{Ghost:} (Ghost A L R W # is, $\theta$, x, m, $D$, $O$, $R$, $S$) $\rightarrow$
  (is, $\theta$, x, m, $D$, $O \cup A - R$, aug (dom $S$) $R$ $R$, $S \oplus_{W R \ominus A L}$)
\textbf{end}


37
**term** direct-memop-step.gen-direct-memop-step

**abbreviation** direct-memop-step :: (instrs × tmeps × unit × memory × bool × owns × rels × shared ) ⇒

(instrs × tmeps × unit × memory × bool × owns × rels × shared ) ⇒ bool

(- → - [60,60] 100)

**where**
direct-memop-step ≡ direct-memop-step.gen-direct-memop-step

**term** x → Y

**abbreviation** direct-memop-steps ::

(instrs × tmeps × unit × memory × bool × owns × unit × shared ) ⇒

(instrs × tmeps × unit × memory × bool × owns × unit × shared ) ⇒ bool

(- →* - [60,60] 100)

**where**
direct-memop-steps == (direct-memop-step)^**

**term** x →* Y


**abbreviation** virtual-memop-step :: (instrs × tmeps × unit × memory × bool × owns × unit × unit × shared ) ⇒

(instrs × tmeps × unit × memory × bool × owns × unit × shared ) ⇒ bool

(- →v - [60,60] 100)

**where**
virtual-memop-step ≡ virtual-memop-step.gen-direct-memop-step

**term** x →v Y

**abbreviation** virtual-memop-steps ::

(instrs × tmeps × unit × memory × bool × owns × unit × shared ) ⇒

(instrs × tmeps × unit × memory × bool × owns × unit × shared ) ⇒ bool

(- →v* - [60,60] 100)

**where**
virtual-memop-steps == (virtual-memop-step)^**

**term** x →v* Y

**lemma** virtual-memop-step-simulates-direct-memop-step:

**assumes** step:

(is, φ, x, m, D, O, R, S) → (is', φ', x', m', D', O', R', S')

**shows** (is, φ, x, m, D, O, (), S) →v (is', φ', x', m', D', O', (), S')

**using** step

**apply** (cases)

**apply** (auto intro: virtual-memop-step.gen-direct-memop-step.intros)
A.4 Safe Configurations of Virtual Machines

\textbf{inductive} safe-direct-memop-state :: owns list ⇒ nat ⇒
\[ (\text{instrs} \times \text{tmps} \times \text{memory} \times \text{owns} \times \text{shared}) ⇒ \text{bool} \]
\[ (-, -) \vdash - [60,60,60] 100 \]
\textbf{where}
Read: 
\[ [a ∈ O \lor a ∈ \text{read-only} S \lor (\text{volatile} \land a ∈ \text{dom} S); \text{volatile} \rightarrow ¬D] \]
\[ ≜ Os,i ⊢ (\text{Read volatile} a \ # \ is, \ θ, m, D, O, S)\]

WriteNonVolatile:
\[ [a ∈ O; a \not\in \text{dom} S] \]
\[ \Rightarrow Os,i ⊢ (\text{Write False} a \ (D,f) A L R W# is, \ θ, m, D, O, S)\]

WriteVolatile:
\[ [∀ j < \text{length} Os. i \neq j \rightarrow a \not\in Os!j; A ⊆ \text{dom} S ∪ O; \text{L} ⊆ A; R \subseteq O; A \cap R = \{\}; ∨ j < \text{length} Os. i \neq j \rightarrow A \cap Os!j = \{\}; a \not\in \text{read-only} S] \]
\[ \Rightarrow Os,i ⊢ (\text{Write True} a \ (D,f) A L R W# is, \ θ, m, D, O, S)\]

Fence:
\[ Os,i ⊢ (\text{Fence} # is, \ θ, m, D, O, S)\]

Ghost:
\[ A \subseteq \text{dom} S ∪ O; \text{L} ⊆ A; R \subseteq O; A \cap R = \{\}; ∨ j < \text{length} Os. i \neq j \rightarrow A \cap Os!j = \{\}; a \not\in \text{read-only} S] \]
\[ \Rightarrow Os,i ⊢ (\text{Ghost} A L R W# is, \ θ, m, D, O, S)\]

RMWReadOnly:
\[ [¬ \text{cond} (\theta(t → m a)); a ∈ O \lor a \in \text{dom} S] \]
\[ \Rightarrow Os,i ⊢ (\text{RMW a} t (D,f) \text{ cond ret} A L R W# is, \ θ, m, D, O, S)\]

RMWWrite:
\[ [\text{cond} (\theta(t → m a)); a \not\in Os!j; A \subseteq \text{dom} S ∪ O; \text{L} ⊆ A; R \subseteq O; A \cap R = \{\}; ∨ j < \text{length} Os. i \neq j \rightarrow A \cap Os!j = \{\}; a \not\in \text{read-only} S] \]
\[ \Rightarrow Os,i ⊢ (\text{RMW a} t (D,f) \text{ cond ret} A L R W# is, \ θ, m, D, O, S)\]

Nil:
\[ Os,i ⊢ ([], \ θ, m, D, O, S)\]
**inductive** safe-delayed-direct-memop-state :: owns list ⇒ rels list ⇒ nat ⇒
(\ \ instrs \times \ tmpps \times \ memory \times \ bool \times \ owns \times \ shared) ⇒ bool
(_\cdot_{\cdot}_{\cdot} \cdot_{\cdot}_{\cdot} \cdot\cdot) \cdot [60,60,60,60] \cdot 100)

**where**
Read: [a ∈ O ∨ a ∈ read-only S ∨ (volatile ∧ a ∈ dom S);
   ∀ j < length Os. i≠j → (Rs!j) a ≠ Some False; (* no release of unshared address *)
   ¬ volatile → (∀ j < length Os. i≠j → a ∉ dom (Rs!j));
   volatile → ¬ D]
⇒
Os,Rs,i−(Read volatile a t # is, φ, m, D, O, S)\v

| WriteNonVolatile: |
[a ∈ O; a ∉ dom S; ∀ j < length Os. i≠j → a ∉ dom (Rs!j)]
⇒
Os,Rs,i−(Write False a (D,f) A L R W#is, φ, m, D, O, S)\v

| WriteVolatile: |
[∀ j < length Os. i≠j → a ∉ (Os!j ∪ dom (Rs!j));
   A ⊆ dom S ∪ O; L ⊆ A; R ⊆ O; A ∩ R = {};
   ∀ j < length Os. i≠j → A ∩ (Os!j ∪ dom (Rs!j)) = {};
   a ∉ read-only S]
⇒
Os,Rs,i−(Write True a (D,f) A L R W# is, φ, m, D, O, S)\v

| Fence: |
Os,Rs,i−(Fence # is, φ, m, D, O, S)\v

| Ghost: |
[A ⊆ dom S ∪ O; L ⊆ A; R ⊆ O; A ∩ R = {};
   ∀ j < length Os. i≠j → A ∩ (Os!j ∪ dom (Rs!j)) = {}]
⇒
Os,Rs,i−(Ghost A L R W# is, φ, m, D, O, S)\v

| RMWReadOnly: |
[¬ cond (θ(t→m a)); a ∈ O ∨ a ∈ dom S;
   ∀ j < length Os. i≠j → (Rs!j) a ≠ Some False (* no release of unshared address *)]
⇒
Os,Rs,i−(RMW a t (D,f) cond ret A L R W# is, φ, m, D, O, S)\v

| RMWrite: |
[cond (θ(t→m a)); a ∈ O ∨ a ∈ dom S;
   ∀ j < length Os. i≠j → a ∉ (Os!j ∪ dom (Rs!j));
   A ⊆ dom S ∪ O; L ⊆ A; R ⊆ O; A ∩ R = {};
   ∀ j < length Os. i≠j → A ∩ (Os!j ∪ dom (Rs!j)) = {};
   a ∉ read-only S]
⇒
Os,Rs,i−(RMW a t (D,f) cond ret A L R W# is, φ, m, D, O, S)\v

| Nil: |
Os,Rs,i−([], φ, m, D, O, S)\v

40
lemma memop-safe-delayed-implies-safe-free-flowing:
  assumes safe-delayed: $O_s,R_s,i \vdash (is, \emptyset, m, D, O, S) \surd$
  shows $O_s,i \vdash (is, \emptyset, m, D, O, S) \surd$
using safe-delayed
proof (cases)
  case Read thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.intros)
next
  case WriteNonVolatile thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.intros)
next
  case WriteVolatile thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.intros)
next
  case Fence thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.intros)
next
  case Ghost thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.Ghost)
next
  case RMWReadOnly thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.RMWWrite)
next
  case RMWWrite thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.RMWWrite)
next
  case Nil thus ?thesis
    by (fastforce intro!: safe-direct-memop-state.Nil)
qed

lemma memop-empty-rels-safe-free-flowing-implies-safe-delayed:
  assumes safe: $O_s,i \vdash (is, \emptyset, m, D, O, S) \surd$
  assumes empty: $\forall R \in \text{set } R_s. \ R = \text{Map.empty}$
  assumes leq: $\text{length } O_s = \text{length } R_s$
  assumes unowned-shared: $(\forall a. (\forall i < \text{length } O_s. a \notin (O_s!i)) \rightarrow a \in \text{dom } S)$
  assumes Os-i: $O_s!i = O$
  shows $O_s,R_s,i \vdash (is, \emptyset, m, D, O, S) \surd$
using safe
proof (cases)
  case Read thus ?thesis
    using leq empty
    by (fastforce intro!: safe-delayed-direct-memop-state.Read dest: nth-mem)
next
  case WriteNonVolatile thus ?thesis
    using leq empty
    by (fastforce intro!: safe-delayed-direct-memop-state.intros dest: nth-mem)
next
  case WriteVolatile thus ?thesis
using leq empty
apply clarsimp
apply (rule safe-delayed-direct-memop-state.WriteVolatile)
apply (auto)
apply (drule nth-mem)
apply fastforce
done
next
case Fence thus ?thesis
  by (fastforce intro!: safe-delayed-direct-memop-state.intros)
next
case Ghost thus ?thesis
  using leq empty
  apply clarsimp
  apply (rule safe-delayed-direct-memop-state.Ghost)
  apply (auto)
  apply (drule nth-mem)
  apply fastforce
done
next
case RMWReadOnly thus ?thesis
  using leq empty unowned-shared [rule-format, where a=a] Os-i
  apply clarsimp
  apply (rule safe-delayed-direct-memop-state.RMWWrite)
  apply (auto)
  apply (drule nth-mem)
  apply fastforce
  apply (drule nth-mem)
  apply fastforce
done
next
case Nil thus ?thesis
  by (fastforce intro!: safe-delayed-direct-memop-state.Nil)
qed

inductive id-storebuffer-step::
(memory × unit × owns × rels × shared) ⇒ (memory × unit × owns × rels × shared)
⇒ bool (- →_1 - [60,60] 100)
where
  Id: (m,x,ɔ,R,ɔ,S) →_1 (m,x,ɔ,R,ɔ)
definition empty-storebuffer-step:: (memory × 'sb × 'owns × 'rels × 'shared) ⇒ (memory × 'sb × 'owns × 'rels × 'shared) ⇒ bool
where
empty-storebuffer-step c c' = False

context program
begin

abbreviation direct-concurrent-step ::
  (p',unit,bool,owns,rels/shared) global-config => (p',unit,bool,owns,rels/shared)
global-config => bool
  (- ⇒₃ [100,60] 100)
where
direct-concurrent-step ≡
  computation.concurrent-step direct-memop-step.gen-direct-memop-step
empty-storebuffer-step program-step
  (λp p' is sb. sb)

abbreviation direct-concurrent-steps::
  (p',unit,bool,owns,rels/shared) global-config => (p',unit,bool,owns,rels/shared)
global-config => bool
  (- ⇒₃ * [60,60] 100)
where
direct-concurrent-steps == direct-concurrent-step`**

abbreviation virtual-concurrent-step ::
  (p',unit,bool,owns,unit/shared) global-config => (p',unit,bool,owns,unit/shared)
global-config => bool
  (- ⇒₄ * [100,60] 100)
where
virtual-concurrent-step ≡
  computation.concurrent-step virtual-memop-step.gen-direct-memop-step
empty-storebuffer-step program-step
  (λp p' is sb. sb)

abbreviation virtual-concurrent-steps::
  (p',unit,bool,owns,unit/shared) global-config => (p',unit,bool,owns,unit/shared)
global-config => bool
  (- ⇒₄ * [60,60] 100)
where
virtual-concurrent-steps == virtual-concurrent-step`**

term x ⇒₄ Y

term x ⇒₃ Y

term x ⇒₄ * Y

term x ⇒₄ * Y

end

definition
safe-reach step safe cfg ≡
\[ \forall \text{cfg}',. \text{step}^{**} \text{cfg} \text{cfg}' \rightarrow \text{safe cfg}' \]

**Lemma safe-reach-safe-refl:** safe-reach step safe \text{cfg} \implies \text{safe cfg}

- **Apply** (auto simp add: safe-reach-def)
- **Done**

**Lemma safe-reach-safe-rtrancl:** safe-reach step safe \text{cfg} \implies \text{step}^{**} \text{cfg} \text{cfg}' \implies \text{safe cfg}'

- **By** (simp only: safe-reach-def)

**Lemma safe-reach-steps:** safe-reach step safe \text{cfg} \implies \text{step}^{**} \text{cfg} \text{cfg}' \implies \text{safe-reach step safe cfg}'

- **Apply** (auto simp add: safe-reach-def intro: rtranclp-trans)
- **Done**

**Lemma safe-reach-step:** safe-reach step safe \text{cfg} \implies \text{step} \text{cfg} \text{cfg}' \implies \text{safe-reach step safe cfg}'

- **Apply** (erule safe-reach-steps)
- **Apply** (erule r-into-rtranclp)
- **Done**

**Context program**

**Begin**

**Abbreviation**

safe-reach-direct \equiv \text{safe-reach direct-concurrent-step}

**Lemma safe-reac-direct-def':**

safe-reach-direct safe \text{cfg} \equiv

- \[ \forall \text{cfg}',. \text{cfg} \Rightarrow_d^{*} \text{cfg}' \rightarrow \text{safe cfg}' \]

- **By** (simp add: safe-reach-def)

**Abbreviation**

safe-reach-virtual \equiv \text{safe-reach virtual-concurrent-step}

**Lemma safe-reac-virtual-def':**

safe-reach-virtual safe \text{cfg} \equiv

- \[ \forall \text{cfg}',. \text{cfg} \Rightarrow_v^{*} \text{cfg}' \rightarrow \text{safe cfg}' \]

- **By** (simp add: safe-reach-def)

**End**

**Definition**

safe-free-flowing \text{cfg} \equiv \text{let } (\text{ts},m,\text{S}) = \text{cfg}

- in \( \forall i < \text{length ts}. \text{let } (p,\text{is},x,D,O,R) = \text{ts}!i \text{ in}

- \text{map owned ts},i \vdash (\text{is},\text{m},D,O,S) \sqrt{)}

**Lemma safeE:** [safe-free-flowing \( (\text{ts},m,\text{S})\); i< length ts; ts!]i \( (\text{ts},\text{m},D,O,R)\)] \implies \text{map owned ts},i \vdash (\text{is},\text{m},D,O,S) \sqrt{)}

- **By** (auto simp add: safe-free-flowing-def)
definition
safe-delayed cfg \equiv \text{let } (ts, m, S) = cfg 
\text{in } (\forall i < \text{length } ts \text{. let } (p, is, \theta, x, D, O, R) = ts!i \text{ in }
\text{map owned } ts, \text{map released } ts, i \vdash (is, \theta, m, D, O, S) \sqrt{})

lemma safe-delayedE: \[\text{safe-delayed } (ts, m, S); i < \text{length } ts; ts!i = (p, is, \theta, x, D, O, R)\]
\[\Rightarrow \text{map owned } ts, \text{map released } ts, i \vdash (is, \theta, m, D, O, S) \sqrt{}\]
by (auto simp add: safe-delayed-def)

definition remove-rels \equiv \text{map } (\lambda (p, is, \theta, sb, D, O, R). (p, is, \theta, sb, D, O, R))

theorem (in program) virtual-simulates-direct-step:
assumes step: \((ts, m, S) \Rightarrow_d (ts', m', S')\)
shows \((\text{remove-rels } ts, m, S) \Rightarrow_v (\text{remove-rels } ts', m', S')\)
using step
proof
- interpret direct-computation:
  computation direct-memop-step empty-storebuffer-step program-step \(\lambda p p' \text{ is sb }\).
- interpret virtual-computation:
  computation virtual-memop-step empty-storebuffer-step program-step \(\lambda p p' \text{ is sb }\).
from step show ?thesis
proof (cases)
case (Program j p is \(\theta \text{ sb }\) \(D \ O \ R \ p' \text{ is}'\))
then obtain
  \(ts': ts' = ts[j := (p', is@is', \theta, sb, D, O, R)]\) and
  \(S': S' = S\) and
  \(m': m' = m\) and
  j-bound: \(j < \text{length } ts\) and
  ts-j: \(ts!j = (p, is, \theta, sb, D, O, R)\) and
  prog-step: \(\theta \vdash p \rightarrow_p (p', is')\)
by auto
from ts-j j-bound have
  vts-j: \(\text{remove-rels } ts!j = (p, is, \theta, sb, D, O, R)\) by (auto simp add: remove-rels-def)

from virtual-computation.Program [OF vts-j prog-step, of m S] j-bound ts'
show ?thesis
by (clarsimp simp add: S' m' remove-rels-def map-update)
next
case (Memop j p is \(\theta \text{ sb }\) \(D \ O \ R \ is' \ \theta' \text{ sb'} D' \ O' \ R'\))
then obtain
  \(ts': ts' = ts[j := (p, is', \theta', sb', D', O', R')]\) and
  j-bound: \(j < \text{length } ts\) and
  ts-j: \(ts!j = (p, is, \theta, sb, D, O, R)\) and
  mem-step: \((is, \theta, sb, m, D, O, R, S) \rightarrow (is', \theta', sb', m', D', O', R', S')\)
by auto
from ts-j j-bound have
  vts-j: \(\text{remove-rels } ts!j = (p, is, \theta, sb, D, O, R)\) by (auto simp add: remove-rels-def)
from virtual-computation.Memop[OF - vts-j]


show ?thesis
  by (clarsimp simp add: remove-rels-def map-update)

next
  case (StoreBuffer - p is \emptyset sb D \cal R sb' \cal O' \cal R')
    hence False
  thus ?thesis ..
  qed

lemmas converse-rtranclp-induct-sbh-steps = converse-rtranclp-induct
[of - (ts,m,S) (ts',m',S'), split-rule,
  consumes 1, case-names refl step]

theorem (in program) virtual-simulates-direct-steps:
  assumes steps: (ts,m,S) \Rightarrow d* (ts',m',S')
  shows (remove-rels ts,m,S) \Rightarrow v* (remove-rels ts',m',S')
  using steps
  proof (induct rule: converse-rtranclp-induct-sbh-steps)
    case refl
    thus ?case by auto
    next
      case (step ts m S ts'' m'' S'')
      then obtain first: (ts,m,S) \Rightarrow d (ts'',m'',S'') and
        hyp: (remove-rels ts'',m'',S'') \Rightarrow v* (remove-rels ts',m',S')
      by blast
      note virtual-simulates-direct-step [OF first]
      also note hyp
    finally
    show ?case by blast
  qed

locale simple-ownership-distinct =

fixes ts::('p,'sb,'dirty,owns,'rels) thread-config list

assumes simple-ownership-distinct:
  \[\forall i j p_i is_i \cal O_i \cal R_i D_i \emptyset_i sb_i is_j \cal O_j \cal R_j D_j \emptyset_j sb_j.
     \[i < \text{length ts}; j < \text{length ts}; i \neq j;
     ts!i = (p_i, is_i, \emptyset_i, sb_i, D_i, \emptyset_i, \cal R_i);
     ts!j = (p_j, is_j, \emptyset_j, sb_j, D_j, \emptyset_j, \cal R_j)
     \] \implies \cal O_i \cap \cal O_j = \{\}\]

lemma (in simple-ownership-distinct)
  simple-ownership-distinct-nth-update:
  \[\forall i p is \emptyset \cal O \cal R D xs is sb.
     \[i < \text{length ts}; ts!i = (p,is,\emptyset, sb,D,\cal O,\cal R);
     \forall j < \text{length ts}. i \neq j \longrightarrow (let (p_i, is_i, \emptyset_i, sb_i, D_i, \cal O_i, \cal R_i) = ts!i
     in (\cal O' \cap (\cal O_j) = \{\})) \implies
     \text{simple-ownership-distinct (ts[i := (p',is',\emptyset',sb',D',\cal O',\cal R')])}\]
  apply (unfold-locales)

46
apply (clarsimp simp add: nth-list-update split: if-split-asm)
apply (force dest: simple-ownership-distinct simp add: Let-def)
apply (fastforce dest: simple-ownership-distinct simp add: Let-def)
done

locale read-only-unowned =
fixes S::shared and ts::('p,'sb,'dirty,owns,'rels) thread-config list
assumes read-only-unowned:
\[\forall i\ p\ is\ ORD\ \theta\ sb.\]
\[\[i < \text{length}\ \text{ts};\ \text{ts}!i = (p,\text{is},\emptyset,\text{sb},\mathcal{O},\mathcal{R})\] \]
\[\Rightarrow\ \mathcal{O} \cap \text{read-only} = \{\}\]

lemma (in read-only-unowned)
read-only-unowned-nth-update:
\[\forall i\ p\ is\ ORD\ \acq\ \theta\ sb.\]
\[\[i < \text{length}\ \text{ts};\ \mathcal{O} \cap \text{read-only} = \{\}\ \Rightarrow\ \text{read-only-unowned} (\text{ts}[i := (p,\text{is},\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R})])\]
apply (unfold-locales)
apply (auto dest: read-only-unowned simp add: nth-list-update split: if-split-asm)
done

locale unowned-shared =
fixes S::shared and ts::('p,'sb,'dirty,owns,'rels) thread-config list
assumes unowned-shared:
\[-\bigcup ((\lambda (-,-,-,\mathcal{O},\mathcal{R})). \mathcal{O}) \ \text{set} \ \text{ts}) \subseteq \text{dom} S\]

lemma (in unowned-shared)
unowned-shared-nth-update:
\[\forall i\ p\ is\ \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \emptyset \ \text{sb}.\]
\[\[i < \text{length}\ \text{ts};\ \mathcal{O} \cap \text{read-only} = \{\}\ \Rightarrow\ \text{read-only-unowned} (\text{ts}[i := (p,\text{is},\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R})])\]
apply (auto simp add: in-set-conv-nth nth-list-update split: if-split-asm)
apply (fastforce simp add: in-set-conv-nth)
done

locale nth-list-update =
fixes S::shared and ts::('p,'sb,'dirty,owns,'rels) thread-config list
assumes nth-list-update:
\[\forall i\ p\ is\ ORD\ \theta\ sb.\]
\[\[i < \text{length}\ \text{ts};\ \text{ts}!i = (p,\text{is},\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R})\] \]
\[\Rightarrow\ \mathcal{O} \cap \text{read-only} = \{\}\]

apply (fastforce dest: simple-ownership-distinct simp add: Let-def)
done

apply (fastforce dest: simple-ownership-distinct simp add: Let-def)
done
hence $- \bigcup \((\lambda(-,-,-,-,\mathcal{O},\cdot)). \mathcal{O}) \setminus \text{set } \{\text{set } \text{ts}[i := (p',is',xs',sb',D',\mathcal{O}',\mathcal{R}')]\} \subseteq$

by blast
also note unowned-shared
finally
show ?thesis
by (unfold-locales)
qed

lemma (in unowned-shared) a-unowned-by-others-owned-or-shared:
assumes i-bound: $i < \text{length ts}$
assumes ts-i: ts!i = (p,is,\theta,sb,D,O,R)
assumes a-unowned-others:
\(\forall j<\text{length (map owned ts)}, i \neq j \rightarrow (\text{let } O_j = (\text{map owned ts})!j \text{ in } a /\in O_j)\)

shows $a \in \mathcal{O} \lor a \in \text{dom } \mathcal{S}$

proof –

\{ fix j p_j is_j O_j R_j D_j xs_j sb_j assume a-unowned: $a /\in \mathcal{O}$ assume j-bound: $j < \text{length ts}$ assume jth: ts!j = (p_j,is_j,xs_j, sb_j, D_j, O_j, R_j) have $a /\in O_j$ proof (cases i=j) case True with a-unowned-others \[\text{rule-format, of } j\] j-bound jth False show ?thesis by auto next from a-unowned-others [rule-format, of j] j-bound jth False show ?thesis by auto qed \}

\} note lem = this
\{
assume a /\in \mathcal{O}
from lem [OF this]
have $a \in - \bigcup \((\lambda(-,-,-,-,\mathcal{O},\cdot)). \mathcal{O}) \setminus \text{set } \text{ts})$
by (fastforce simp add: in-set-conv-nth)
with unowned-shared have $a \in \text{dom } \mathcal{S}$
by auto
\}
then show ?thesis by auto
qed

lemma (in unowned-shared) unowned-shared':
assumes notin: $\forall i < \text{length ts}. a /\in \text{owned } (\text{ts}[i])$
shows \( a \in \text{dom} \, S \)

proof –
from notin have \( a \in - \bigcup ((\lambda (\_ , \_ , \_ , \_ , \_ , \_ , \_ , O , O) , O) : \text{set} \, ts) \)
by (force simp add: in-set-conv-nth)
with unowned-shared show \(?\text{thesis} \) by blast

qed

lemma unowned-shared-def': unowned-shared \( S \) \( ts = (\forall a. (\forall i < \text{length} \, ts. a \notin \text{owned} (ts!i)) \rightarrow a \in \text{dom} \, S) \)
apply rule
apply clarsimp
apply (rule unowned-shared.unowned-shared')
apply fastforce
apply fastforce
apply (unfold unowned-shared-def)
apply clarsimp
subgoal for \( x \)
apply (drule-tac \( x=x \) in spec)
apply (erule impE)
apply clarsimp
apply (case-tac (ts!i))
apply (drule nth-mem)
apply auto
done
done

definition initial cfg \( \equiv \) let \( (ts,m,S) = cfg \)
in unowned-shared \( S \) \( ts \land \)
(\( \forall i < \text{length} \, ts. \) let \( (p,\text{is},\theta,x,D,O,R) = ts!i \) in
\( R = \text{Map.empty} \) )

lemma initial-empty-rels: initial \( (ts,m,S) \) \( \Longrightarrow \forall R \in \text{set} \, (\text{map} \, \text{released} \, ts). R = \text{Map.empty} \)
by (fastforce simp add: initial-def simp add: in-set-conv-nth)

lemma initial-unowned-shared: initial \( (ts,m,S) \) \( \Longrightarrow \) unowned-shared \( S \) \( ts \)
by (fastforce simp add: initial-def )

lemma initial-safe-free-flowing-implies-safe-delayed:
assumes init: initial \( c \)
assumes safe: safe-free-flowing \( c \)
shows safe-delayed \( c \)
proof –
obtain \( ts \) \( S \) \( m \) where \( c=(ts,m,S) \) by (cases \( c \) )
from initial-empty-rels [OF init [simplified \( c \)]]
have rels-empty: \( \forall R\in\text{set} \, (\text{map} \, \text{released} \, ts). R = \text{Map.empty} . \)
from initial-unowned-shared [OF init [simplified \( c \)]] have unowned-shared \( S \) \( ts \)
by auto
hence us: (\( \forall a. (\forall i < \text{length} \, (\text{map} \, \text{owned} \, ts) . a \notin (\text{map} \, \text{owned} \, ts!i)) \rightarrow a \in \text{dom} \, S) \)

49
by (simp add: unowned-shared-def)
{
  fix i p is δ x D O R
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,δ,x,D,O,R)
  have map owned ts,map released ts,i ⊢ (is,δ,m,D,O,S) ∨
  proof
    - from safeE [OF safe [simplified c] i-bound ts-i]
      have map owned ts,i ⊢ (is,δ,m,D,O,S) ∨
    from memop-empty-rels-safe-free-flowing-implies-safe-delayed [OF this rels-empty - us] i-bound ts-i
      show ?thesis
      by simp
    qed
  }
  then show ?thesis
  by (fastforce simp add: c safe-delayed-def)
qed

locale program-progress = program +
assumes progress: δ ⊢ p → p (p′,is′) ⇒ p′ ≠ p ∨ is′ ≠ []
  The assumption ‘progress’ could be avoided if we introduce stuttering steps in lemma
undo-local-step or make the scheduling of threads explicit, such that we can directly express
that ‘thread i does not make a step’.lemma (in program-progress) undo-local-step:
assumes step: (ts,m,S) ⇒ d (ts′,m′,S′)
assumes i-bound: i < length ts
assumes unchanged: ts!i = ts′!i
assumes safe-delayed-undo: safe-delayed (u-ts,u-m,u-shared) — proof should also work
with weaker safe-free-flowing
assumes leq: length u-ts = length ts
assumes others-same: ∀ j < length ts. j ≠ i → u-ts!j = ts!j
assumes u-ts-i: u-ts!i=(u-p,u-is,u-tmps,u-x,u-dirty,u-owns,u-rels)
assumes u-m-other: ∀ a. a /∈ u-owns → u-m a = m a
assumes u-m-shared: ∀ a. a ∈ u-owns → a ∈ dom u-shared → u-m a = m a
assumes u-shared: ∀ a. a /∈ u-owns → a /∈ owned (ts!i) → u-shared a = S a
assumes dist: simple-ownership-distinct u-ts
assumes dist-ts: simple-ownership-distinct ts
shows ∃ u-ts′ u-shared′ u-m′. (u-ts,u-m,u-shared) ⇒d (u-ts′,u-m′,u-shared′) ∧
  — thread i is unchanged
  u-ts′!i = u-ts!i ∧
  (∀ a ∈ u-owns. u-shared′ a = u-shared a) ∧
  (∀ a ∈ u-owns. S′ a = S a) ∧
  (∀ a ∈ u-owns. u-m′ a = u-m a) ∧
  (∀ a ∈ u-owns. m′ a = m a) ∧
  — other threads are simulated
  (∀ j < length ts. j ≠ i → u-ts′!j = ts′!j) ∧
  (∀ a. a /∈ u-owns → a /∈ owned (ts!i) → u-shared′ a = S′ a) ∧
  (∀ a. a /∈ u-owns → u-m′ a = m′ a)
proof

interpret direct-computation:
computation direct-memop-step empty-storebuffer-step program-step \( \lambda p' \) is sb. sb.

from dist interpret simple-ownership-distinct u-ts.

from step
show ?thesis

proof (cases)
case (Program j p is \( \emptyset \) sb \( D \ O \ R \) p' is')
then obtain
ts': \( ts[j:=(p',is@is',\emptyset,\emptyset,\emptyset,D,O,R)] \) and
S': \( S' = S \) and
m': \( m' = m \) and
j-bound: \( j < \text{length } ts \) and
ts-j: \( ts[j] = (p,0,0,\emptyset,D,O,R) \) and
prog-step: \( \emptyset \vdash p \rightarrow_p (p', is') \)
by auto

from progress [OF prog-step] i-bound unchanged ts-j ts'
have neq-j-i: \( j \neq i \)
by auto

from others-same [rule-format, OF j-bound neq-j-i] ts-j
have u-ts-j: \( u-ts[j] = (p,0,0,\emptyset,D,O,R) \)
by simp
from leq j-bound have j-bound': \( j < \text{length } u-ts \)
by simp
from leq i-bound have i-bound': \( i < \text{length } u-ts \)
by simp

from direct-computation.Program [OF j-bound' u-ts-j prog-step]
have ustep: \( (u-ts,u-m, u-shared) \Rightarrow_d (u-ts[j := (p', is @ is', \emptyset, \emptyset, D, O, R)], u-m, u-shared) \).
show ?thesis
apply
apply (rule exI)
apply (rule exI)
apply (rule exI)
apply (rule conjI)
apply (rule ustep)
using neq-j-i others-same u-m-other u-shared j-bound leq ts-j
apply (auto simp add: nth-list-update ts' S' m')
done

next
case (Memop j p is \( \emptyset \) sb \( D \ O \ R \) is' \( \emptyset \) sb' \( D' \ O' \ R' \))
then obtain
ts': \( ts[j:=(p,is',\emptyset,sb',D',O',R')] \) and
j-bound: \( j < \text{length } ts \) and
ts-j: \( ts[j] = (p,0,0,\emptyset,D',O',R) \) and
mem-step: \( (is, \emptyset, sb, m, D, O, R, S) \rightarrow (is', \emptyset, sb', m', D', O', R', S') \)
by auto
from mem-step i-bound unchanged ts-j
have neq-j-i: j≠i
  by cases (auto simp add: ts′)

from others-same [rule-format, OF j-bound neq-j-i] ts-j
have u-ts-j: u-ts!j = (p, is, δ, sb, D, O, R)
  by simp
from leq j-bound have j-bound': j < length u-ts
  by simp
from leq i-bound have i-bound': i < length u-ts
  by simp
from safe-delayedE [OF safe-delayed-undo j-bound' u-ts-j]
have safe-j: map owned u-ts,map released u-ts,j⁻→(is, δ, u-m, D, O, u-shared)√.
from simple-ownership-distinct [OF j-bound' i-bound' neq-j-i u-ts-j u-ts-i]
have owns-u-owns: O ∩ u-owns = {}. from mem-step
show ?thesis
proof (cases)
case (Read volatile a t)
then obtain
  is: is = Read volatile a t ≠ is' and
  δ': δ' = δ(t ↦→ m a) and
  sb': sb = sb and
  m': m = m and
  D': D = D and
  O': O = O and
  R': R = R and
  S': S = S
  by auto
note eqs' = δ' sb' m' D' O' R' S'

from safe-j [simplified is]
obtain
  access-cond: a ∈ O ∨ a ∈ read-only u-shared ∨
  (volatile ∧ a ∈ dom u-shared)
  and
  clean: volatile → ¬ D
  by cases auto
have u-m-a-eq: u-m a = m a
proof (cases a ∈ u-owns)
case True
  with simple-ownership-distinct [OF j-bound' i-bound' neq-j-i u-ts-j u-ts-i]
  have a ≠ O by auto
  with access-cond read-only-dom [of u-shared] have a ∈ dom u-shared
    by auto
  from u-m-shared [rule-format, OF True this]
  show ?thesis .
next
case False
from u-m-other [rule-format, OF this]
show ?thesis.

direct-memop-step.Read [of volatile a t is' \theta sb u-m D O R u-shared]
from direct-computation.Memop [OF j-bound' u-ts-j, simplified is, OF Read']

next

next case (WriteNonVolatile a D f A L R W)
then obtain
is: is = Write False a (D, f) A L R W # is' and
\theta': \theta' = \theta and
sb': sb'=sb and
m': m'=m(a:=f \theta) and
D': D'=D and
O': O'=O and
R': R'=R and
S': S'=S
by auto

next case (WriteNonVolatile a D f A L R W)
then obtain
is: is = Write False a (D, f) A L R W # is' and
\theta': \theta' = \theta and
sb': sb'=sb and
m': m'=m(a:=f \theta) and
D': D'=D and
O': O'=O and
R': R'=R and
S': S'=S
by auto

from safe-j [simplified is]

obtain
owned: a \in O and unshared: a \notin \text{dom} u-shared
by cases auto

from simple-ownership-distinct [OF j-bound' i-bound' neq-j-i u-ts-j u-ts-i] owned

have a-unowned-i: a \notin u-owns
by auto

next case (WriteNonVolatile a D f A L R W)
then obtain
is: is = Write False a (D, f) A L R W # is' and
\theta': \theta' = \theta and
sb': sb'=sb and
m': m'=m(a:=f \theta) and
D': D'=D and
O': O'=O and
R': R'=R and
S': S'=S
by auto
note eqs' = \theta sb' m' D' O' R' S'
by auto

from safe-j [simplified is]

obtain
owned: a \in O and unshared: a \notin \text{dom} u-shared
by cases auto
apply (rule ustep)
using neq-j-i others-same u-m-other u-shared j-bound leq ts-j a-unowned-i
apply (auto simp add: nth-list-update ts′ eqs′)
done

next
case (WriteVolatile a D f A L R W)
then obtain
is: is = Write True a (D, f) A L R W # is′ and
θ′; θ′ = θ and
sb′; sb′=sb and
m′; m′=m(a:=f θ) and
D′; D′=True and
O′; O′=O ∪ A − R and
R′; R′=Map.empty and
S′; S′=S ⊕ W R ⊕ A L
by auto
note eqs′ = θ′ sb′ m′ D′ O′ R′ S′

from safe-j [simplified is]
obtain
a-unowned-others: ∀k < length u-ts. j≠k → a /∈ (map owned u-ts!k ∪ dom (map released u-ts!k)) and
A: A ⊆ dom u-shared ∪ O and L-A: L ⊆ A and R-owns: R ⊆ O and A-R: A ∩ R = {} and
A-unowned-others: ∀k < length u-ts. j≠k → A ∩ (map owned u-ts!k ∪ dom (map released u-ts!k)) = {} and
a-not-ro: a /∈ read-only u-shared
by cases auto

note Write′ = direct-memop-step.WriteVolatile [of a D f A L R W is′ θ sb u-m D O R u-shared]
from direct-computation.Memop [OF j-bound′ u-ts-j, simplified is, OF Write′]
have ustep: (u-ts, u-m, u-shared) ⇒d (u-ts[j := (p, is′, θ, sb, True, O ∪ A − R, Map.empty)], u-m (a := f θ), u-shared ⊕ W R ⊕ A L).

from A-unowned-others [rule-format, OF i-bound′ neq-j-i] u-ts-i i-bound′
have A-u-owns: A ∩ u-owns = {} by auto
{
fix a
assume a-u-owns: a ∈ u-owns
have (u-shared ⊕ W R ⊕ A L) a = u-shared a
using R-owns A-R L-A A-u-owns owns-u-owns a-u-owns
by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)
}
note u-owned-shared = this
from a-unowned-others [rule-format, OF i-bound′ neq-j-i] u-ts-i i-bound′ have
a-u-owns: a /∈ u-owns by auto
{
fix a

54
\textbf{assume} a-u-owns: a \not\in u-owns
\textbf{assume} a-u-owns-orig: a \not\in owned (ts!i)
\textbf{from} u-shared [rule-format, OF a-u-owns a-u-owns-orig]
\textbf{have} (u-shared \oplus W R \ominus_A L) a = (S \oplus W R \ominus_A L) a
\textbf{using} R-owns A-R L-A A-u-owns owns-u-owns
\textbf{by} (auto simp add: restrict-shared-def augment-shared-def split: option.splits)
\}
\textbf{note} u-unowned-shared = this
\{ 
\textbf{fix} a
\textbf{assume} a-u-owns: a \in u-owns
\textbf{have} (S \oplus W R \ominus_A L) a = S a
\textbf{using} R-owns A-R L-A A-u-owns owns-u-owns
\textbf{by} (auto simp add: restrict-shared-def augment-shared-def split: option.splits)
\}
\textbf{note} S\textsuperscript{'}-shared = this
\textbf{show} \texttt{?thesis}
\textbf{apply} –
\textbf{apply} (rule exI)
\textbf{apply} (rule exI)
\textbf{apply} (rule conjI)
\textbf{apply} (rule ustep)
\textbf{using} neq-j-i others-same u-m-other u-shared j-bound leq ts-j u-owned-shared a-u-owns u-unowned-shared S\textsuperscript{'}-shared
\textbf{apply} (auto simp add: nth-list-update ts\textsuperscript{'} eqs\textsuperscript{'}
\textbf{done}
\textbf{next}
\textbf{case} Fence
\textbf{then obtain}
\textbf{is:} is = Fence \# is\textsuperscript{'} \textbf{and}
\textbf{\emptyset':} \emptyset' = \emptyset \textbf{and}
\textbf{sb':} sb' = sb \textbf{and}
\textbf{m':} m' = m \textbf{and}
\textbf{D':} D' = False \textbf{and}
\textbf{O':} O' = O \textbf{and}
\textbf{R':} R' = Map.empty \textbf{and}
\textbf{S':} S' = S
\textbf{by} auto
\textbf{note} eqs\textsuperscript{'} = \emptyset' sb' m' D' O' R' S'
\textbf{note} Fence' = direct-memop-step.Fence [of is\textsuperscript{'} \emptyset sb u-m D O R u-shared]
\textbf{from} direct-computation.Memop [OF j-bound' u-ts-j, simplified is, OF Fence']
\textbf{have} ustep: (u-ts, u-m, u-shared) \Rightarrow_d (u-ts[j := (p, is\textsuperscript{'} , \emptyset, sb, False, O, Map.empty)], u-m, u-shared).
\textbf{show} \texttt{?thesis}
\textbf{apply} –
\textbf{apply} (rule exI)
\textbf{apply} (rule exI)

55
apply (rule exI)
apply (rule conjI)
apply (rule ustep)
using neq-j-i others-same u-m-other u-shared j-bound leq ts-j
by (auto simp add: nth-list-update ts′ eqs′)

next
case (RMWReadOnly cond t a D f ret A L R W)
then obtain
is: is = RMW a t (D, f) cond ret A L R W # is' and
θ': θ' = θ(t → m a) and
sb': sb'="sb" and
m': m'=m and
D': D'=False and
O': O'=O and
R': R'=Map.empty and
S': S'=S and
cond: ¬ cond (θ(t → m a))
by auto
note eqs' = θ' sb' m' D' O' R' S'
from safe-j [simplified is] owns-u-owns u-ts-i i-bound' neq-j-i
obtain
access-cond: a ∉ u-owns ∨ (a ∈ dom u-shared ∧ a ∈ u-owns)
by cases auto

from u-m-other u-m-shared access-cond
have u-m-a-eq: u-m a = m a
by auto
from cond u-m-a-eq have cond': ¬ cond (θ(t → u-m a))
by auto
note RMWReadOnly' = direct-memop-step.RMWReadOnly [of cond θ t u-m a D f ret A L R W is' sb D O R u-shared, OF cond']
from direct-computation.Memop [OF j-bound' u-ts-j, simplified is, OF RMWReadOnlyOnly']
have ustep: (u-ts, u-m, u-shared) ⇒d (u-ts[j := (p, is', θ(t → u-m a), sb, False, O, Map.empty)], u-m, u-shared).
show ?thesis
apply
apply (rule exI)
apply (rule exI)
apply (rule exI)
apply (rule conjI)
apply (rule ustep)
using neq-j-i others-same u-m-other u-shared j-bound leq ts-j
by (auto simp add: nth-list-update ts′ eqs′ u-m-a-eq)

next
case (RMWWrite cond t a D f ret A L R W)
then obtain
is: is = RMW a t (D, f) cond ret A L R W # is' and
θ': θ' = θ(t → ret (m a) (f (θ(t → m a)))) and

56
\( \text{sb}' \colon \text{sb}' \equiv \text{sb} \quad \text{and} \quad \text{m}' \colon \text{m}' = \text{m}(a := f (\emptyset(t \mapsto m a))) \quad \text{and} \quad \mathcal{D}' \colon \mathcal{D}' = \text{False} \quad \text{and} \quad \mathcal{O}' \colon \mathcal{O}' = \mathcal{O} \cup \mathcal{A} - \mathcal{R} \quad \text{and} \quad \mathcal{R}' \colon \mathcal{R}' = \text{Map.empty} \quad \text{and} \quad \mathcal{S}' \colon \mathcal{S}' = \mathcal{S} \oplus \mathcal{W} \mathcal{R} \ominus \mathcal{A} \mathcal{L} \quad \text{and} \quad \text{cond}: \text{cond} (\emptyset(t \mapsto m a)) \\
\text{by auto} \)

\textbf{note} eqs' = \( \emptyset' \text{sb}' \text{m}' \mathcal{D}' \mathcal{O}' \mathcal{R}' \mathcal{S}' \)

\textbf{from} safe-j [simplified is] owns-u-owns u-ts-i i-bound' neq-j-i

\textbf{obtain}

access-cond: \( a \notin \text{u-owns} \lor (a \in \text{dom u-shared} \land a \in \text{u-owns}) \)

\text{by cases auto} \)

\textbf{from} u-m-other u-m-shared access-cond

\textbf{have} u-m-a-eq: u-m a = m a

\text{by auto} \)

\textbf{from} cond u-m-a-eq \textbf{have} cond': cond (\emptyset(t \mapsto u-m a))

\text{by auto} \)

\textbf{from} safe-j [simplified is] cond'

\textbf{obtain}

a-unowned-others: \( \forall k < \text{length u-ts}. \, j \neq k \rightarrow a \notin (\text{map owned u-ts}!k \cup \text{dom (map released u-ts}!k)) \) \text{ and }

A: A \subseteq \text{dom u-shared} \cup \mathcal{O} \quad \text{and} \quad L-A: L \subseteq A \quad \text{and} \quad \text{R-owns: R} \subseteq \mathcal{O} \quad \text{and} \quad \text{A-R: A} \cap \mathcal{R} = \{\} \quad \text{and} 

\text{A-unowned-others: } \forall k < \text{length u-ts}. \, j \neq k \rightarrow A \cap (\text{map owned u-ts}!k \cup \text{dom (map released u-ts}!k)) = \{\} \quad \text{and} 

\text{a-not-ro: } a \notin \text{read-only u-shared}

\text{by cases auto} \)

\textbf{note} Write' = direct-memop-step.RMWWrite [of cond \emptyset t u-m a D f ret A L R W is' sb D O R u-shared, 

\text{OF cond}]

\textbf{from} direct-computation.Memop [OF j-bound' u-ts-j, simplified is, OF Write']

\textbf{have} ustep: u-ts\colon (u-ts, u-m, u-shared) \Rightarrow_d 

(\emptyset(t \mapsto \text{ret (u-m a)} (f (\emptyset(t \mapsto u-m a)))), \text{sb, False, } \mathcal{O} \cup \mathcal{A} - \mathcal{R}, \text{Map.empty}])], u-m(a := f (\emptyset(t \mapsto u-m a))), \text{u-shared } \ominus \mathcal{W} \mathcal{R} \ominus \mathcal{A} \mathcal{L}).

\textbf{from} A-unowned-others [rule-format, OF i-bound' neq-j-i] u-ts-i i-bound'

\textbf{have} A-u-owns: A \cap \text{u-owns} = \{\} \quad \text{by auto}

\{ 

\text{fix} a 

\text{assume} a-u-owns: a \in \text{u-owns}

\textbf{have} (u-shared \ominus \mathcal{W} \mathcal{R} \ominus \mathcal{A} \mathcal{L}) \, a = \text{u-shared a}

\textbf{using} \text{R-owns A-R L-A} A-u-owns owns-u-owns a-u-owns

\text{by} (\text{auto simp add: restrict-shared-def augment-shared-def split: option.splits})

\}

\textbf{note} u-owned-shared = this

57
from a-unowned-others [rule-format, OF i-bound’ neq-j-i] u-ts-i i-bound’ have a-u-owns: a \notin u-owns by auto

{ fix a
 assume a-u-owns: a \notin u-owns
 assume a-u-owns-orig: a \notin owned (ts!i)
 from u-shared [rule-format, OF a-u-owns this]
 have (u-shared \oplus_W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a
 using R-owns A-R L-A A-u-owns owns-u-owns
 by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)
 }
 note u-unowned-shared = this

{ fix a
 assume a-u-owns: a \in u-owns

 have (S \oplus_W R \ominus_A L) a = S a
 using R-owns A-R L-A A-u-owns owns-u-owns a-u-owns
 by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)
 }
 note S'-shared = this

show ?thesis
 apply –
 apply (rule exI)
 apply (rule exI)
 apply (rule exI)
 apply (rule conjI)
 apply (rule ustep)
     using neq-j-i others-same u-m-other u-shared j-bound leq ts-j u-owned-shared
 a-u-owns u-unowned-shared S'-shared
 apply (auto simp add: nth-list-update ts' eqs')
 done
 next
 case (Ghost A L R W)
 then obtain
 is: is = Ghost A L R W # is' and
 \emptyset': \emptyset' = \emptyset and
 sb': sb' = sb and
 m': m' = m and
 D': D' \subseteq D and
 O': O' = O \cup A - R and
 R': R' = augment-rels (dom S) R R and
 S': S' = S \oplus_W R \ominus_A L
 by auto
 note eqs' = \emptyset' sb' m' D' O' R' S'

 from safe-j [simplified is]
 obtain
 A: A \subseteq dom u-shared \cup O and L-A: L \subseteq A and R-owns: R \subseteq O and A-R: A \cap R
 = {} and
\[ \forall k < \text{length } u-ts. \ j \neq k \rightarrow A \cap (\text{map owned } u-ts!k \cup \text{dom (map released } u-ts!k)) = \{\} \]

by cases auto

\textbf{note} Ghost’ = direct-memop-step.Ghost [of A L R W is’ \emptyset sb u-m \ D \ O \ R u-shared]

\textbf{from} direct-computation.Memop [OF j-bound’ u-ts-j, simplified is, OF Ghost’]

\textbf{have} ustep: (u-ts, u-m, u-shared) \Rightarrow_d

\[ (u-ts[j := (p, is’, \emptyset, sb, D, O \cup A - R, \text{augment-rels (dom u-shared) R R })], u-m, \]

\[ \text{u-shared } \oplus_W R \ominus_A L). \]

\textbf{from} A-unowned-others [rule-format, OF i-bound’ neq-j-i] u-ts-i i-bound’

\textbf{have} A-u-owns: A \cap u-owns = \{\} by auto

\{ \}

\textbf{fix} a

\textbf{assume} a-u-owns: a \in u-owns

\textbf{have} (u-shared \ominus_W R \ominus_A L) a = u-shared a

\textbf{using} R-owns A-R L-A A-u-owns owns-u-owns a-u-owns

by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)

\}

\textbf{note} u-owned-shared = this

\{ \}

\textbf{fix} a

\textbf{assume} a-u-owns: a \notin u-owns

\textbf{assume} a \notin owned (ts!i)

\textbf{from} u-shared [rule-format, OF a-u-owns this]

\textbf{have} (u-shared \ominus_W R \ominus_A L) a = (S \ominus_W R \ominus_A L) a

\textbf{using} R-owns A-R L-A A-u-owns owns-u-owns

by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)

\}

\textbf{note} u-unowned-shared = this

\{ \}

\textbf{fix} a

\textbf{assume} a-u-owns: a \in u-owns

\textbf{have} (S \ominus_W R \ominus_A L) a = S a

\textbf{using} R-owns A-R L-A A-u-owns owns-u-owns a-u-owns

by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)

\}

\textbf{note} S’-shared = this

\textbf{from} dist-ts

\textbf{interpret} dist-ts-inter: simple-ownership-distinct ts .

\textbf{from} dist-ts-inter.simple-ownership-distinct [OF j-bound i-bound neq-j-i ts-j]

\textbf{have} O \cap owned (ts!i) = \{\} \]

\textbf{apply} (cases ts!i)

\textbf{apply} fastforce+

\textbf{done}
with simple-ownership-distinct [OF j-bound' i-bound' neq-j-i u-ts-j u-ts-i] R-owns u-shared

have augment-eq: augment-rels (dom u-shared) R R = augment-rels (dom S) R R
apply –
apply (rule ext)
apply (fastforce simp add: augment-rels-def split: option.splits simp add: domIff)
done

show ?thesis
apply –
apply (rule exI)
apply (rule exI)
apply (rule exI)
apply (rule conjI)
apply (rule ustep)
using neq-j-i others-same u-m-other u-shared j-bound leq ts-j u-owned-shared u-unowned-shared S'-shared
apply (auto simp add: nth-list-update ts' eqs' augment-eq)
done
qed
next
case (StoreBuffer - p is \(\emptyset\) sb D O R sb' O' R')
hence False
by (auto simp add: empty-storebuffer-step-def)
thus ?thesis ..
qed
qed

theorem (in program) safe-step-preserves-simple-ownership-distinct:
assumes step: \((ts,m,S) \Rightarrow_d (ts',m',S')\)
assumes safe: safe-delayed \((ts,m,S)\)
assumes dist: simple-ownership-distinct ts
shows simple-ownership-distinct ts'
proof –
interpret direct-computation:
computation direct-memop-step empty-storebuffer-step program-step \(\lambda p \quad p'\) is sb. sb .
from dist interpret simple-ownership-distinct ts .
from step
show ?thesis
proof (cases)
case (Program j p is \(\emptyset\) sb D O R p' is')
then obtain
ts': \(ts' = ts[j := (p', is@is', \emptyset, sb, D, O, R)]\) and
S': \(S' = S\) and
m': \(m' = m\) and
j-bound: \(j < \text{length} \ ts\) and
ts-j: \(ts!j = (p, is, \theta, sb, D, O, R)\) and
prog-step: \(\theta \vdash p \rightarrow p'(p', is')\)
by auto

from simple-ownership-distinct [OF j-bound - - ts-j]
show simple-ownership-distinct ts'
apply (simp only: ts')
apply (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])
apply force
done
next
case (Memop j p is \(\theta\) sb DOR is' \(\theta'\) sb' D' O' R')
then obtain
\(ts'': ts'' = ts[j:=(p, is', \theta', sb', D', O', R')]\) and
j-bound: \(j < \text{length} ts\) and
ts-j: \(ts!j = (p, is, sb, D, O, R)\) and
mem-step: \((is, \theta, sb, m, D, O, S) \rightarrow (is', \theta', sb', m', D', O', R', S')\)
by auto

from safe-delayedE [OF safe j-bound ts-j]
have safe-j: \(\text{map} \text{owned} ts, \text{map} \text{released} ts, j \vdash (is, \theta, m, D, O, S)\).√.
from mem-step
show \(?\)thesis
proof (cases)
case (Read volatile a t)
then obtain
is: is = Read volatile a t # is' and
\(\theta': \theta' = \theta(t \mapsto m a)\) and
sb': sb'=sb and
m': m'=m and
D': D'=D and
O': O'=O and
R': R'=R and
S': S'=S
by auto
note eqs' = \(\theta' sb' m' D' O' R' S'\)

from simple-ownership-distinct [OF j-bound - - ts-j]
show simple-ownership-distinct ts'
apply (simp only: ts' eqs')
apply (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])
apply force
done
next
case (WriteNonVolatile a D f A L R W)
then obtain
is: is = Write False a (D, f) A L R W # is' and
\(\theta': \theta' = \theta\) and
sb': sb'=sb and
m': m'=m(a:=f \emptyset) \text{ and}
D': D'=D \text{ and}
O': O'=O \text{ and}
R': R'=R \text{ and}
S': S'=S
by auto

\textbf{note} eqs' = \emptyset' sb' m' D' O' R' S'
\textbf{from} simple-ownership-distinct [OF j-bound - ts-j]
\textbf{show} simple-ownership-distinct ts'
\hspace{1em} \textbf{apply} (simp only: ts' eqs')
\hspace{1em} \textbf{apply} (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])
\hspace{1em} \textbf{apply} force
\hspace{1em} \textbf{done}

\textbf{next}
\textbf{case} (WriteVolatile a D f A L R W)
\textbf{then obtain}
\hspace{1em} is: is = Write True a (D, f) A L R W # is' \text{ and}
\hspace{1em} \emptyset': \emptyset' = \emptyset \text{ and}
\hspace{1em} sb': sb'=sb \text{ and}
\hspace{1em} m': m'=m(a:=f \emptyset) \text{ and}
D': D'=True \text{ and}
O': O'=O \cup A - R \text{ and}
R': R'=Map.empty \text{ and}
S': S'=S \oplus W R \ominus A L
\hspace{1em} by auto

\textbf{note} eqs' = \emptyset' sb' m' D' O' R' S'
\textbf{from} safe-j [simplified is]
\textbf{obtain}
\hspace{1em} a-unowned-others: \forall k < \text{length ts}. \ j\neq k \rightarrow a \not\in (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) \text{ and}
A: A \subseteq \text{dom S} \cup O \text{ and L-A: L} \subseteq A \text{ and R-owns: R} \subseteq O \text{ and A-R: A} \cap R = \{\}
\hspace{1em} and
\hspace{1em} A-unowned-others: \forall k < \text{length ts}. \ j\neq k \rightarrow A \cap (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) = \{\} \text{ and}
\hspace{1em} a-not-ro: a \not\in \text{read-only S}
\hspace{1em} by cases auto
\textbf{show} simple-ownership-distinct ts'
\hspace{1em} \textbf{apply} (simp only: ts' eqs')
\hspace{1em} \textbf{apply} (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])
\hspace{1em} \textbf{apply} force
\hspace{1em} \textbf{done}

\textbf{next}
\textbf{case} Fence
\textbf{then obtain}
\hspace{1em} is: is = Fence # is' \text{ and}
\hspace{1em} \emptyset': \emptyset' = \emptyset \text{ and}
\hspace{1em} sb': sb'=sb \text{ and}
m': m' = m and
D': D' = False and
O': O' = O and
R': R' = Map.empty and
S': S' = S
by auto
note eqs' = \dot{\varnothing}' \cdot sb' \cdot m' \cdot D' \cdot O' \cdot R' \cdot S'
from simple-ownership-distinct [OF j-bound - - ts-j]
show simple-ownership-distinct ts'
  apply (simp only: ts' eqs')
  apply (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])
  apply force
  done
next
case (RMWReadOnly cond t a D f ret A L R W)
then obtain
  is: is = RMW a t (D, f) cond ret A L R W \# is' and
  \dot{\varnothing}': \dot{\varnothing}' = \dot{\varnothing}(t \mapsto m a) and
  sb': sb' = sb and
  m': m' = m and
  D': D' = False and
  O': O' = O and
  R': R' = Map.empty and
  S': S' = S and
  cond: \neg cond (\dot{\varnothing}(t \mapsto m a))
  by auto
note eqs' = \dot{\varnothing}' \cdot sb' \cdot m' \cdot D' \cdot O' \cdot R' \cdot S'
from simple-ownership-distinct [OF j-bound - - ts-j]
show simple-ownership-distinct ts'
  apply (simp only: ts' eqs')
  apply (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])
  apply force
  done
next
case (RMWWrite cond t a D f ret A L R W)
then obtain
  is: is = RMW a t (D, f) cond ret A L R W \# is' and
  \dot{\varnothing}': \dot{\varnothing}' = \dot{\varnothing}(t \mapsto ret (m a) (f (\dot{\varnothing}(t \mapsto m a)))) and
  sb': sb' = sb and
  m': m' = m(a := f (\dot{\varnothing}(t \mapsto m a))) and
  D': D' = False and
  O': O' = O \cup A - R and
  R': R' = Map.empty and
  S': S' = S \oplus W R \ominus A L and
  cond: cond (\dot{\varnothing}(t \mapsto m a))
  by auto
note eqs' = \dot{\varnothing}' \cdot sb' \cdot m' \cdot D' \cdot O' \cdot R' \cdot S'
from safe-j [simplified is] cond
obtain
a-unowned-others: \( \forall k < \text{length ts}. \; j \neq k \rightarrow a \notin (\text{map owned ts!}k \cup \text{dom (map released ts!}k)) \) \textbf{and} \\
A: A \subseteq \text{dom } S \cup O \textbf{ and } L-A: L \subseteq A \textbf{ and } R-owns: R \subseteq O \textbf{ and } A-R: A \cap R = \{\}

and

A-unowned-others: \( \forall k < \text{length ts}. \; j \neq k \rightarrow A \cap (\text{map owned ts!}k \cup \text{dom (map released ts!}k)) = \{\} \) \textbf{and} \\
a-not-ro: a \notin \text{read-only } S

by cases auto

\textbf{from} simple-ownership-distinct \( \text{[OF j-bound - - ts-j]} \) R-owns A-R A-unowned-others \\
\textbf{show} simple-ownership-distinct ts'

\textbf{apply} (simp only: ts' eqs')

\textbf{apply} (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])

\textbf{apply} force

done

next

case (Ghost A L R W)

\textbf{then obtain} \\
is: is = Ghost A L R W # is' \textbf{and} \\
\emptyset': \emptyset' = \emptyset \textbf{ and} \\
sb': sb'=sb \textbf{ and} \\
m': m'=m \textbf{ and} \\
D': D'=D \textbf{ and} \\
O': O'=O \cup A - R \textbf{ and} \\
R': R'=\text{augment-rels (dom } S) R \textbf{ and} \\
S': S'=S \oplus W R \oplus_A L \\
by auto

\textbf{note} eqs' = \emptyset' sb' m' D' O' R' S'

\textbf{from} safe-j [simplified is] \\
\textbf{obtain} \\
A: A \subseteq \text{dom } S \cup O \textbf{ and } L-A: L \subseteq A \textbf{ and } R-owns: R \subseteq O \textbf{ and } A-R: A \cap R = \{\}

and

A-unowned-others: \( \forall k < \text{length ts}. \; j \neq k \rightarrow A \cap (\text{map owned ts!}k \cup \text{dom (map released ts!}k)) = \{\} \)

by cases auto

\textbf{from} simple-ownership-distinct \( \text{[OF j-bound - - ts-j]} \) R-owns A-R A-unowned-others \\
\textbf{show} simple-ownership-distinct ts'

\textbf{apply} (simp only: ts' eqs')

\textbf{apply} (rule simple-ownership-distinct-nth-update [OF j-bound ts-j])

\textbf{apply} force

done

\textbf{qed}

next

case (StoreBuffer - p is \emptyset D O R sb' O' R')

\textbf{hence} False

\textbf{by} (auto simp add: empty-storebuffer-step-def)

\textbf{thus} \textit{?thesis} ..

\textbf{qed}
theorem (in program) safe-step-preserves-read-only-unowned:
  assumes step: (ts,m,S) ⇒ (ts',m',S')
  assumes safe: safe-delayed (ts,m,S)
  assumes dist: simple-ownership-distinct ts
  assumes ro-unowned: read-only-unowned S ts
shows read-only-unowned S' ts'
proof
  interpret direct-computation:
    computation direct-memop-step empty-storebuffer-step program-step λp p' is sb. sb .
  from interpret simple-ownership-distinct ts .
  from ro-unowned interpret read-only-unowned S ts .
  from step
show ?thesis
proof (cases)
case (Program j p is θ sb D O R p' is')
  then obtain
    ts': ts' = ts[j:=(p',is@is',θ,Σd,D,O,R)] and
    S': S'=S and
    m': m'=m and
  j-bound: j < length ts and
  ts-j: tsj = (p,is,θ,Σd,D,O,R) and
  prog-step: θ⊢ p → p (p',is')
  by auto
from read-only-unowned [OF j-bound ts-j]
show read-only-unowned S' ts'
  apply (simp only: ts' S')
  apply (rule read-only-unowned-nth-update [OF j-bound])
  apply force
  done
next
case (Memop j p is θ sb D O R is' θ' sb' D' O' R')
  then obtain
    ts': ts' = ts[j:=(p,is,θ,Σd,D',O',R')] and
  j-bound: j < length ts and
  ts-j: tsj = (p,is,θ,Σd,D,O,R) and
  mem-step: (is, θ, sb, m, D, O, R, S) → (is', θ', sb',m', D', O', R', S')
  by auto
from safe-delayedE [OF safe j-bound ts-j]
have safe-j: map owned ts,map released ts, j (is, θ, m, D, O, S)√.
from mem-step
show ?thesis
proof (cases)
case (Read volatile a t)
  then obtain
    is: is = Read volatile a t ≠ is' and
    θ': θ' = θ(t → m a) and

qed
sb': sb' = sb and
m': m' = m and
D': D' = D and
O': O' = O and
R': R' = R and
S': S' = S
by auto
note eqs' = θ' sb' m' D' O' R' S'

from read-only-unowned [OF j-bound ts-j]
show read-only-unowned S' ts'
  apply (simp only: ts' eqs')
  apply (rule read-only-unowned-nth-update [OF j-bound])
  apply force
done

next
case (WriteNonVolatile a D f A L R W)
then obtain
  is: is = Write False a (D, f) A L R W # is' and
  θ' θ' = θ and
  sb': sb' = sb and
  m': m' = m(a:=f θ) and
  D': D' = D and
  O': O' = O and
  R': R' = R and
  S': S' = S
by auto
note eqs' = θ' sb' m' D' O' R' S'
from read-only-unowned [OF j-bound ts-j]
show read-only-unowned S' ts'
  apply (simp only: ts' eqs')
  apply (rule read-only-unowned-nth-update [OF j-bound])
  apply force
done

next
case (WriteVolatile a D f A L R W)
then obtain
  is: is = Write True a (D, f) A L R W # is' and
  θ' θ' = θ and
  sb': sb' = sb and
  m': m' = m(a:=f θ) and
  D': D' = True and
  O': O' = O ∪ A − R and
  R': R' = Map.empty and
  S' S' = S ⊕ W R ⊕ A L
by auto
note eqs' = θ' sb' m' D' O' R' S'
from safe-j [simplified is]

obtain
  a-unowned-others: \( \forall k < \text{length } ts \), \( j \neq k \rightarrow a \notin (\text{map owned } ts!k \cup \text{dom (map released } ts!k)) \) and
  A: A \( \subseteq \text{dom } S \cup O \) and L-A: L \( \subseteq A \) and R-owns: R \( \subseteq O \) and A-R: A \cap R = \{ \} and
  a-unowned-others: \( \forall k < \text{length } ts \), \( j \neq k \rightarrow A \cap (\text{map owned } ts!k \cup \text{dom (map released } ts!k)) = \{ \} \) and
  a-not-ro: a \notin \text{read-only } S
by cases auto

show read-only-unowned \( S' \) \( ts' \)
proof (unfold-locale)
fix i p i is_i O_i D_i \( \theta_i \) sb_i
assume i-bound: i < length \( ts' \)
assume ts'\( ^{i}\): ts'\( ^{i}\) = (p, \( \text{is} \)\( _i \), \( \theta_i \), sb_i, \( D_i \), O_i, R_i)
show \( O_i \cap \text{read-only } S' = \{ \} \)
proof (cases \( i=j \))
case True
with read-only-unowned [OF j-bound ts-j] \( ts'\)\( ^{i}\) A L-A R-owns A-R j-bound
show \?thesis
by (auto simp add: \( eqs' \) \( ts' \) read-only-def augment-shared-def restrict-shared-def domIff split: option.splits)
next
case False
from simple-ownership-distinct [OF j-bound False [symmetric] ts-j] \( ts'\)\( ^{i}\) i-bound
j-bound False
have \( O \cap O_i = \{ \} \)
by (fastforce simp add: \( ts' \)')
with A L-A R-owns A-R j-bound A-unowned-others [rule-format, of i]
read-only-unowned [of i \( p_i \) \( \text{is} \)\( _i \) \( \theta_i \) sb_i \( D_i \) O_i R_i]
False i-bound \( ts'\)\( ^{i}\) False
show \?thesis
by (force simp add: 
(\( eqs' \) \( ts' \) read-only-def augment-shared-def restrict-shared-def domIff split: option.splits)
qed
qed
next
case Fence
then obtain
is: is = Fence \# is' and
\( \theta' \): \( \theta' = \theta \) and
sb': sb' = sb and
m': m' = m and
\( D' \): D' = False and
\( O' \): O' = O and
\( R' \): R' = Map.empty and
\( S' \): S' = S
by auto


note eqs' = \( \theta' \) sb' \( m' \) D' O' R' S'
from read-only-unowned [OF j-bound ts-j]
show read-only-unowned $S'$
apply (simp only: ts' eqs')
apply (rule read-only-unowned-nth-update [OF j-bound])
apply force
done
next
case (RMWReadOnly cond t a D f ret A L R W)
then obtain
is: is = RMW a t (D, f) cond ret A L R W # is' and
$\theta'$: $\theta' = \theta(t \mapsto m_a)$ and
sb': sb' = sb and
m': m' = m and
$D'$: $D' = \text{False}$ and
$O'$: $O' = O$ and
$R'$: $R' = \text{Map.empty}$ and
$S'$: $S' = S$ and
cond: $\neg \text{cond (}\theta(t \mapsto m_a))$
by auto
note eqs' = $\theta'$ sb' m' $D'$ $O'$ $R'$ $S'$
from read-only-unowned [OF j-bound ts-j]
show read-only-unowned $S'$
apply (simp only: ts' eqs')
apply (rule read-only-unowned-nth-update [OF j-bound])
apply force
done
next
case (RMWWrite cond t a D f ret A L R W)
then obtain
is: is = RMW a t (D, f) cond ret A L R W # is' and
$\theta'$: $\theta' = \theta(t \mapsto \text{ret (m_a) (f (}\theta(t \mapsto m_a))))$ and
sb': sb' = sb and
m': m' = m(a := f (\theta(t \mapsto m_a))) and
$D'$: $D' = \text{False}$ and
$O'$: $O' = O \cup A - R$ and
$R'$: $R' = \text{Map.empty}$ and
$S'$: $S' = S \oplus W R \ominus A L$ and
cond: $\text{cond (}\theta(t \mapsto m_a))$
by auto
note eqs' = $\theta'$ sb' m' $D'$ $O'$ $R'$ $S'$
from safe-j [simplified is] cond obtain
a-unowned-others: $\forall k < \text{length ts. } j \neq k \rightarrow a \notin (\text{map owned ts!k } \cup \text{ dom (map released ts!k))}$ and
A: $A \subseteq \text{dom } S \cup O$ and L-A: $L \subseteq A$ and R-owns: $R \subseteq O$ and A-R: $A \cap R = \{\}$ and
A-unowned-others: $\forall k < \text{length ts. } j \neq k \rightarrow A \cap (\text{map owned ts!k } \cup \text{ dom (map released ts!k))} = \{\}$ and
a-not-ro: $a \notin \text{read-only } S$
by cases auto

68
show read-only-unowned $S'$ ts'

proof (unfold-locales)
  fix i p i s $O_i$ $R_i$ $D_i$ $\emptyset_i$ sb $i$
  assume i-bound: $i < \text{length } ts'$
  assume ts'':i: ts'':i = ($p_i$, is $i$, $\emptyset_i$, sb $i$, $D_i$, $O_i$, $R_i$)
  show $O_i \cap \text{read-only } S'$ = {}
  proof (cases i=j)
    case True
    with read-only-unowned [OF j-bound ts-j] ts'':i A L-A R-owns A-R j-bound
    show ?thesis
    by (auto simp add: eqs' ts' read-only-def augment-shared-def restrict-shared-def domIff split: option.splits)
  next
    case False
    from simple-ownership-distinct [OF j-bound - False [symmetric] ts-j] ts'':i i-bound False
    have $O \cap O_i$ = {}
    by (fastforce simp add: ts'')
    with A L-A R-owns A-R j-bound A-unowned-others [rule-format, of i]
    read-only-unowned [of i p i is $i$ $\emptyset_i$ sb $i$ $D_i$ $O_i$ $R_i$]
    False i-bound ts'':i False
    show ?thesis
    by (force simp add: eqs' ts' read-only-def augment-shared-def restrict-shared-def domIff split: option.splits)
  qed
  qed

next
  case (Ghost A L R W)
  then obtain
    is: is = Ghost A L R W # is' and
    $\emptyset'$: $\emptyset' = \emptyset$ and
    sb': sb' = sb and
    m': m' = m and
    $D'$: $D' = D$ and
    $O'$: $O' = O \cup A - R$ and
    $R'$: $R' = \text{augment-rels (dom } S) R R$ and
    $S'': S'' = S \oplus W R \ominus A L$
    by auto
  note eqs' = $\emptyset'$ sb' m' $D'$ $O'$ $R'$ $S''$

  from safe-j [simplified is]
  obtain
    A: A $\subseteq$ dom $S \cup O$ and
    L-A: L $\subseteq A$ and
    R-owns: R $\subseteq O$ and
    A-R: A $\cap R = \{}$
    and
    A-unowned-others: $\forall k < \text{length } ts. j \neq k \rightarrow A \cap (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) = \{}$
    by cases auto

  show read-only-unowned $S'$ ts'

69
proof (unfold-locales)
fix i p_is D_i v_i sb_i
assume i-bound: i < length ts'
assume ts'-i: ts'!i = (p_is, D_i, O!, R!, i)
show O! ∩ read-only S' = {}
proof (cases i=j)
case True
with read-only-unowned [OF j-bound ts-j] ts'-i A L-A R-owns A-R j-bound
show ?thesis
  by (auto simp add: eqs' ts' read-only-def augment-shared-def restrict-shared-def
    domIff split: option.splits)
next
case False
from simple-ownership-distinct [OF j-bound - False j-bound ts-j] ts'-i i-bound
have O∩O_i = {}
  by (fastforce simp add: ts')
with A L-A R-owns A-R j-bound A-unowned-others [rule-format, of i]
  read-only-unowned [of i p_is v_i sb_i D_i O_i R_i]
  False i-bound ts'-i False
show ?thesis
  by (force simp add: eqs' ts' read-only-def augment-shared-def restrict-shared-def
    domIff split: option.splits)
qed
next
case (StoreBuffer - p is θ sb D OR sb' O'R')
hence False
  by (auto simp add: empty-storebuffer-step-def)
thus ?thesis ..
qed
qed

next
assume step: (ts,m,S) ⇒_d (ts',m',S')
assume safe: safe-delayed (ts,m,S)
assume dist: simple-ownership-distinct ts
assume unowned-shared: unowned-shared S ts
shows unowned-shared S' ts'
proof -
interpret direct-computation:
  computation direct-memop-step empty-storebuffer-step program-step λp p' is sb. sb .
from dist interpret simple-ownership-distinct ts .
from unowned-shared interpret unowned-shared S ts .
from step show ?thesis
proof (cases)
case (Program j p is θ sb D OR p' is')

70
then obtain
\[ t's' \; t's' = t's[j := (p', is'@is', \emptyset, sb, D, O, R)] \; \text{and} \]
\[ S' : S'=S \; \text{and} \]
\[ m' : m'=m \; \text{and} \]
j-bound: \( j < \text{length } t's \; \text{and} \]
\[ t's-j: t's!j = (p, is, \emptyset, sb, D, O, R) \; \text{and} \]
prog-step: \( \emptyset \vdash p \to_p (p', is') \)
by auto

show unowned-shared \( S' \; t's' \)
apply (simp only: \( t's' \) )
apply (rule unowned-shared-nth-update [OF j-bound ts-j])
apply force
done

next
case (Memop j p is \( \emptyset \) sb D O R is' \( \emptyset \) sb' D' O' R')
then obtain
\[ t's' \; t's' = t's[j := (p,is',\emptyset, sb', D', O', R')]] \; \text{and} \]
j-bound: \( j < \text{length } t's \; \text{and} \]
\[ t's-j: t's!j = (p,is,\emptyset, sb, D, O, R) \; \text{and} \]
mem-step: \( (is, \emptyset, sb, m, D, O, S) \to (is', \emptyset, sb', m', D', O', R', S') \)
by auto
from safe-delayedE [OF safe j-bound ts-j]
have safe-j: map owned ts,map released ts,j-\( t's, (is, \emptyset, m, D, O, S) \) √.
from mem-step
show ?thesis
proof (cases)
case (Read volatile a t)
then obtain
is: is = Read volatile a t # is' \; \text{and} \]
\( \emptyset' : \emptyset' = \emptyset (t \mapsto m a) \; \text{and} \]
\( sb' : sb' = sb \; \text{and} \]
m': \( m'=m \; \text{and} \]
\( D' : D'=D \; \text{and} \]
\( O' : O'=O \; \text{and} \]
\( R' : R'=R \; \text{and} \]
\( S' : S'=S \)
by auto
note eqs' = \( \emptyset' sb' m' D' O' R' S' \)
show unowned-shared \( S' \; t's' \)
apply (simp only: \( t's' \; eqs' \) )
apply (rule unowned-shared-nth-update [OF j-bound ts-j])
apply force
done

next
case (WriteNonVolatile a D f A L R W)
then obtain
is: is = Write False a (D, f) A L R W # is' and
\[ \theta; \theta' = \emptyset \text{ and} \]
\[ s'b; sb'=sb \text{ and} \]
\[ m'; m'=m(a:=f \emptyset) \text{ and} \]
\[ D'; D'=D \text{ and} \]
\[ O'; O'=O \text{ and} \]
\[ R'; R'=R \text{ and} \]
\[ S'; S'=S \]
\[ \text{by auto} \]
\[ \text{note eqs'} = \emptyset' sb' m' D' O' R' S' \]

\[ \text{show unowned-shared } S' \text{ ts'} \]
\[ \text{apply (simp only: ts' eqs')} \]
\[ \text{apply (rule unowned-shared-nth-update [OF j-bound ts-j])} \]
\[ \text{apply force} \]
\[ \text{done} \]

next
\[ \text{case (WriteVolatile a D f A L R W)} \]
\[ \text{then obtain} \]
\[ \text{is: is = Write True a (D, f) A L R W # is' and} \]
\[ \emptyset; \emptyset' = \emptyset \text{ and} \]
\[ s'b; sb'=sb \text{ and} \]
\[ m'; m'=m(a:=f \emptyset) \text{ and} \]
\[ D'; D'=True \text{ and} \]
\[ O'; O'=O \cup A - R \text{ and} \]
\[ R'; R'=Map.empty \text{ and} \]
\[ S'; S'=S \oplus W R \ominus A L \]
\[ \text{by auto} \]
\[ \text{note eqs'} = \emptyset' sb' m' D' O' R' S' \]

from safe-j [simplified is]
\[ \text{obtain} \]
\[ \text{a-unowned-others: } \forall k < \text{ length ts}. j\neq k \longrightarrow a \notin (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) \text{ and} \]
\[ A: A \subseteq \text{ dom } S \cup O \text{ and } L-A: L \subseteq A \text{ and } R-owns: R \subseteq O \text{ and } A-R: A \cap R = \{ \} \text{ and} \]
\[ \text{A-unowned-others: } \forall k < \text{ length ts}. j\neq k \longrightarrow A \cap (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) = \{ \} \text{ and} \]
\[ a-not-ro: a \notin \text{ read-only } S \]
\[ \text{by cases auto} \]

\[ \text{show unowned-shared } S' \text{ ts'} \]
\[ \text{apply (clarsimp simp add: unowned-shared-def')} \]
\[ \text{using A R-owns L-A A-R A-unowned-others ts-j j-bound} \]
\[ \text{apply (auto simp add: S' ts' O')} \]
\[ \text{apply (rule unowned-shared')} \]
\[ \text{apply clarsimp} \]
\[ \text{apply (drule-tac x=i in spec)} \]

72
apply (case-tac i=j)
apply clarsimp
apply clarsimp
apply (drule-tac x=j in spec)
apply auto
done

next
case Fence
then obtain
  is: is = Fence # is' and
  \( \emptyset' \) = \( \emptyset \) and
  sb': sb'=sb and
  m': m'=m and
  D': D'=False and
  O': O'=O and
  R': R'=Map.empty and
  S': S'=S
by auto
note eqs' = \( \emptyset' \) sb' m' D' O' R' S'

show unowned-shared S' ts'
apply (simp only: ts' eqs')
apply (rule unowned-shared-nth-update [OF j-bound ts-j])
apply force
done

next
case (RMWReadOnly cond t a D f ret A L R W)
then obtain
  is: is = RMW a t (D, f) cond ret A L R W # is' and
  \( \emptyset' \) = \( \emptyset(t \mapsto m \ a) \) and
  sb': sb'=sb and
  m': m'=m and
  D': D'=False and
  O': O'=O and
  R': R'=Map.empty and
  S': S'=S and
  cond: \( \neg \) cond (\( \emptyset(t \mapsto m \ a) \))
by auto
note eqs' = \( \emptyset' \) sb' m' D' O' R' S'
show unowned-shared S' ts'
apply (simp only: ts' eqs')
apply (rule unowned-shared-nth-update [OF j-bound ts-j])
apply force
done

next
case (RMWWrite cond t a D f ret A L R W)
then obtain
  is: is = RMW a t (D, f) cond ret A L R W # is' and
  \( \emptyset' \) = \( \emptyset(t \mapsto ret (m \ a) (f (\emptyset(t \mapsto m \ a)))) \) and
  sb': sb'=sb and
m': m' := m(a := f (θ(t → m a))) and
D': D' := D and
O': O' := O ∪ A - R and
R': R' := Map.empty and
S': S' := S ⊕ W R ⊕ A L and
cond: cond (θ(t → m a)) by auto
note eqs' = θ' sb' m' D' O' R' S'
from safe-j [simplified is] cond
obtain
a-unowned-others: ∀ k < length ts. j ≠ k → a ∉ (map owned ts[k] ∪ dom (map released ts[k])) and
and
A-unowned-others: ∀ k < length ts. j ≠ k → A ∩ (map owned ts[k] ∪ dom (map released ts[k])) = {} and
a-not-ro: a ∉ read-only S
by cases auto

show unowned-shared S' ts'
apply (clarsimp simp add: unowned-shared-def')
using A R-owns L-A A-R A-unowned-others ts-j j-bound
apply (auto simp add: S' ts' O')
apply (rule unowned-shared')
apply clarsimp
apply (drule-tac x=i in spec)
apply (case-tac i=j)
apply clarsimp
apply clarsimp
apply (drule-tac x=j in spec)
apply auto
done
next
case (Ghost A L R W)
then obtain
is: is = Ghost A L R W # is' and
θ': θ' = θ and
sb': sb' = sb and
m': m' = m and
D': D' = D and
O': O' = O ∪ A - R and
R': R' = Map.empty and
S': S' = S ⊕ W R ⊕ A L
by auto
note eqs' = θ' sb' m' D' O' R' S'
from safe-j [simplified is]
obtain
A-unowned-others: \( \forall k < \text{length ts}. j \neq k \rightarrow A \cap (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) = \{\} \)

by cases auto

show unowned-shared \( S' \) ts

apply (clarsimp simp add: unowned-shared-def')

using A R-owns L-A A-R A-unowned-others ts-j j-bound

apply (auto simp add: \( S' \) ts' \( O' \))

apply (rule unowned-shared')

apply clarsimp

apply (drule-tac x=i in spec)

apply (clarsimp simp add: clarsimp)

apply clarsimp

apply clarsimp

apply (drule-tac x=j in spec)

apply (clarsimp simp add: spec)

apply (rule unowned-shared')

apply clarsimp

apply clarsimp

apply (drule-tac x=j in spec)

apply auto

done

qed

next

case (StoreBuffer - p is \( \theta \) sb \( D \cup R \) sb' \( O' \) \( R' \))

hence False

by (auto simp add: empty-storebuffer-step-def)

thus \?thesis ..

qed

qed

locale program-trace = program +

fixes c — enumeration of configurations: \( c \ n \Rightarrow_d c \ (n + 1) \ldots \Rightarrow_d c \ (n + k) \)

fixes \( n::\text{nat} \) — starting index

fixes \( k::\text{nat} \) — steps

assumes step: \( \forall l. l < k \rightarrow c \ (n+l) \Rightarrow_d c \ (n + (\text{Suc} \ l)) \)

abbreviation (in program)
trace \equiv program-trace program-step

lemma (in program) trace-0 [simp]: trace c n 0

apply (unfold-locales)

apply auto

done

lemma split-less-Suc: \( \forall x<k. k \ x = \ (P \ k \land (\forall x<k. \ P \ x)) \)

apply rule

apply clarsimp

apply clarsimp

apply (case-tac x = k)

apply auto

done

lemma split-le-Suc: \( \forall x\leq k. k \ x = \ (P \ (\text{Suc} \ k) \land (\forall x\leq k. \ P \ x)) \)

apply rule
apply clarsimp
apply clarsimp
apply (case-tac x = Suc k)
apply auto
done

lemma (in program) steps-to-trace:
assumes steps: x ⇒\text{*}y
shows \(\exists c,k. \text{trace } c \ 0 \ k \land c \ 0 = x \land c \ k = y\)
using steps
proof (induct)
  case base
  thus ?case
    apply (rule-tac x=λk. x in exI)
    apply (rule-tac x=0 in exI)
    by (auto simp add: program-trace-def)
next
  case (step y z)
  have first: x ⇒\text{*}y by fact
  have last: y ⇒\text{*}z by fact
  from step.hyps obtain c k where
    trace: \text{trace } c \ 0 \ k \ \text{and } c \ 0 = x \ \text{and } c \ k = y
    by auto
  define c' where c' == λi. (if i ≤ k then c i else z)
  from trace last c-k have trace c' 0 (k + 1)
    apply (clarsimp simp add: c'-def program-trace-def)
    apply (simp)
    apply (simp)
    done
  with c-0
  show ?case
    apply −
    apply (rule-tac x=c' in exI)
    apply (rule-tac x=k + 1 in exI)
    apply (auto simp add: c'-def)
    done
qed

lemma (in program) trace-preserves-length-ts:
\(\forall \ x. \text{trace } c \ n \ k \Rightarrow l \leq k \Rightarrow x \leq k \Rightarrow \text{length } (\text{fst } (c \ (n + l))) = \text{length } (\text{fst } (c \ (n + x)))\)
proof (induct k)
  case 0
  thus ?case by auto
next
  case (Suc k)
  then obtain trace-suc: \text{trace } c \ n \ (Suc k) \ \text{and}
    l-suc: l ≤ Suc k \ \text{and}
x-suc: $x \leq \text{Suc } k$
by simp
interpret direct-computation:
computation direct-memop-step empty-storebuffer-step program-step $\lambda p p'$ is sb. sb.

from trace-suc obtain
trace-k: trace $c \; n \; k$ and
last-step: $c \; (n + k) \Rightarrow_d c \; (n + (\text{Suc } k))$
by (clarsimp simp add: program-trace-def)
obtain $ts \; S \; m$ where c-k: $c \; (n + k) = (ts, m, S)$ by (cases $c \; (n + k)$)
obtain $ts' \; S' \; m'$ where c-suc-k: $c \; (n + (\text{Suc } k)) = (ts', m', S')$ by (cases $c \; (n + (\text{Suc } k))$)
from direct-computation.step-preserves-length-ts [OF last-step [simplified c-k c-suc-k]]
c-k c-suc-k
have leq: length (fst (c (n + Suc k))) = length (fst (c (n + k)))
by simp

show ?case
proof (cases $l = \text{Suc } k$)
case True
note l-suc = this
show ?thesis
proof (cases $x = \text{Suc } k$)
case True with l-suc show ?thesis by simp
next
case False
with x-suc have $x \leq k$ by simp
from Suc.hyps [OF trace-k this, of k]
have length (fst (c (n + x))) = length (fst (c (n + k)))
by simp
with leq show ?thesis using l-suc by simp
qed
next
case False
with l-suc have l-k: $l \leq k$
by auto
show ?thesis
proof (cases $x = \text{Suc } k$)
case True
from Suc.hyps [OF trace-k l-k, of k]
have length (fst (c (n + l))) = length (fst (c (n + k))) by simp
with leq True show ?thesis by simp
next
case False
with x-suc have $x \leq k$ by simp
from Suc.hyps [OF trace-k l-k this]
show ?thesis by simp
qed
qed
qed
lemma (in program) trace-preserves-simple-ownership-distinct:
assumes dist: simple-ownership-distinct (fst (c n))
shows \( \forall l. \text{trace c n k} \implies (\forall x < k. \text{safe-delayed} (c (n + x))) \implies l \leq k \implies \text{simple-ownership-distinct} (fst (c (n + l))) \)

proof (induct k)
case 0 thus ?case using dist by auto
next
case (Suc k)
then obtain
  trace-suc: trace c n (Suc k) and
  safe-suc: \( \forall x < \text{Suc k}. \text{safe-delayed} (c (n + x)) \) and
  l-suc: \( l \leq \text{Suc k} \)
  by simp

from trace-suc obtain
  trace-k: trace c n k and
  last-step: \( c (n + k) \implies c (n + (\text{Suc k})) \)
  by (clarsimp simp add: program-trace-def)

obtain ts S m where c-k: \( c (n + k) = (ts, m, S) \) by (cases c (n + k))
obtain ts' S' m' where c-suc-k: \( c (n + (\text{Suc k})) = (ts', m', S') \) by (cases c (n + (Suc k)))

from safe-suc c-suc-k c-k
go
take safe-up-k: \( \forall x < k. \text{safe-delayed} (c (n + x)) \) and
safe-k: \( \text{safe-delayed} (ts, m, S) \)
  by (auto simp add: split-le-Suc)
from Suc.hyps [OF trace-k safe-up-k]
have hyp: \( \forall l \leq k. \text{simple-ownership-distinct} (fst (c (n + l))) \)
  by simp

from Suc.hyps [OF trace-k safe-up-k, of k] c-k
have simple-ownership-distinct ts
  by simp

from safe-step-preserves-simple-ownership-distinct [OF last-step[simplified c-k c-suc-k]
safe-k this]
  have simple-ownership-distinct ts'.
then show ?case
  using c-suc-k hyp l-suc
  apply (cases l=Suc k)
  apply (auto simp add: split-less-Suc)
done
qed

lemma (in program) trace-preserves-read-only-unowned:
assumes dist: simple-ownership-distinct (fst (c n))
assumes ro: read-only-unowned (snd (snd (c n))) (fst (c n))
shows $\forall l. \text{trace } c \ n \ k \implies (\forall x < k. \text{safe-delayed} (c (n + x))) \implies 
\begin{align*}
1 \leq k & \implies \text{read-only-unowned} (\text{snd} (\text{snd} (c (n + l)))) (\text{fst} (c (n + l)))
\end{align*}$

proof (induct $k$

case 0 thus ?case using ro by auto

next

case (Suc $k$)

then obtain

trace-suc: $\text{trace } c \ n \ (\text{Suc } k)$ and

safe-suc: $\forall x < \text{Suc } k. \text{safe-delayed} (c (n + x))$ and

l-suc: $1 \leq \text{Suc } k$

by simp

from trace-suc obtain

trace-k: $\text{trace } c \ n \ k$ and

last-step: $c (n + k) \Rightarrow_{d} c (n + (\text{Suc } k))$

by (clarsimp simp add: program-trace-def)

obtain $\text{ts } S \ m$ where $c \ (n + k) = (\text{ts}, \ m, S)$ by (cases $c \ (n + k)$)

obtain $\text{ts' } S' \ m'$ where $c-\text{Suc-}k: c \ (n + (\text{Suc } k)) = (\text{ts'}, \ m', S')$ by (cases $c \ (n + (\text{Suc } k))$)

from safe-suc c-suc-k c-k obtain

safe-up-k: $\forall x < k. \text{safe-delayed} (c (n + x))$ and

safe-k: $\text{safe-delayed} \ (\text{ts}, \ m, S)$

by (auto simp add: split-le-Suc)

from Suc.hyps [OF trace-k safe-up-k]

have hyp: $\forall l \leq k. \text{read-only-unowned} (\text{snd} (\text{snd} (c (n + l)))) (\text{fst} (c (n + l)))$

by simp

from Suc.hyps [OF trace-k safe-up-k, of $k$] c-k

have ro': $\text{read-only-unowned} \ S' \ ts$

by simp

from trace-preserves-simple-ownership-distinct [where $c=c$ and $n=n$, OF dist trace-k safe-up-k, of $k$] c-k

have dist': $\text{simple-ownership-distinct} \ \text{ts}$ by simp

from safe-step-preserves-read-only-unowned [OF last-step[simplified c-k c-suc-k] safe-k dist' ro']

have $\text{read-only-unowned} \ S' \ ts'$.

then show ?case

using c-suc-k hyp l-suc

apply (cases $l=\text{Suc } k$)

apply (auto simp add: split-less-Suc)

done

qed

lemma (in program) trace-preserves-unowned-shared:

assumes dist: $\text{simple-ownership-distinct} \ (\text{fst} (c \ n))$
assumes ro: unowned-shared (snd (snd (c n))) (fst (c n))
shows \( \bigwedge \lambda l. \text{trace } c \ n \ k \Rightarrow (\forall x < k. \text{safe-delayed } (c \ (n + x))) \Rightarrow l \leq k \Rightarrow \text{unowned-shared } (\text{snd } (\text{snd } (c \ (n + l)))) (\text{fst } (c \ (n + l))) \)

proof (induct k)
case 0 thus ?case using ro by auto
next
case (Suc k)
then obtain
  trace-suc: trace c n (Suc k) and
  safe-suc: \( \forall x < \text{Suc } k. \text{safe-delayed } (c \ (n + x)) \) and
  l-suc: \( l \leq \text{Suc } k \)
  by simp

from trace-suc obtain
  trace-k: trace c n k and
  last-step: \( c \ (n + k) \Rightarrow_d c \ (n + (\text{Suc } k)) \)
  by (clarsimp simp add: program-trace-def)

obtain ts \( S \) m where c-k: \( c \ (n + k) = (ts, m, S) \) by (cases \( c \ (n + k) \))
obtain ts' \( S' \) m' where c-suc-k: \( c \ (n + (\text{Suc } k)) = (ts', m', S') \) by (cases \( c \ (n + (\text{Suc } k)) \))

from safe-suc c-suc-k c-k
obtain
  safe-up-k: \( \forall x < k. \text{safe-delayed } (c \ (n + x)) \) and
  safe-k: safe-delayed (ts,m,S)
  by (auto simp add: split-le-Suc)
from Suc.hyps [OF trace-k safe-up-k]
have hyp: \( \forall l \leq k. \text{unowned-shared } (\text{snd } (\text{snd } (c \ (n + l)))) (\text{fst } (c \ (n + l))) \)
  by simp

from Suc.hyps [OF trace-k safe-up-k, of k] c-k
have ro': unowned-shared \( S \) ts
  by simp

from trace-preserves-simple-ownership-distinct [where c=c and n=n, OF dist trace-k safe-up-k, of k] c-k
have dist': simple-ownership-distinct ts by simp

from safe-step-preserves-unowned-shared [OF last-step[simplified c-k c-suc-k] safe-k dist']
  ro'
have unowned-shared \( S' \) ts'.
then show ?case
using c-suc-k hyp l-suc
  apply (cases l=Suc k)
  apply (auto simp add: split-less-Suc)
done
qed
theorem (in program-progress) undo-local-steps:

assumes steps: trace c n k
assumes c-n: c n = (ts,m,S)
assumes unchanged: ∀ l ≤ k. (∀ ts_1 m_1, c (n + l) = (ts_1,m_1,S_1) → ts_l!i=ts_l)
assumes safe: safe-delayed (u-ts, u-m, u-shared)
assumes leq: length u-ts = length ts
assumes i-bound: i < length ts
assumes others-same: ∀ j < length ts. j = i → u-ts[j] = ts[j]
assumes u-ts-i: u-ts[i]=(u-p,u-is,u-tmps,u-sb,u-dirty,u-owns,u-rels)
assumes u-m-other: ∀ a. a ∉ u-owns → u-m a = m a
assumes u-m-shared: ∀ a. a ∈ u-owns → a ∈ dom u-shared → u-m a = m a
assumes u-shared: ∀ a. a ∉ u-owns → owned (ts[i]) → u-shared a = S a
assumes dist: simple-ownership-distinct u-ts
assumes dist-ts: simple-ownership-distinct ts
assumes safe-orig: ∀ x. x < k → safe-delayed (c (n + x))

shows ∃ c'n l. l ≤ k ∧ trace c'n l ∧

(∀ x ≤ l. length (fst (c' (n + x)))) = length (fst (c (n + x)))) ∧

(∀ x < l. length (fst (c' (n + x)))) ∧
(1 < k → ¬ safe-delayed (c' (n + l)))) ∧

(∀ x ≤ l. ∀ ts_x S_x m_x ts'_x S'_x m'_x. c (n + x) = (ts_x,m_x,S_x) → c' (n+ x) =
(ts'_x,m'_x,S'_x) →

(∀ a ∈ u-owns. S'_x a = u-shared a) ∧
(∀ a ∈ u-owns. m'_x a = u-m a) ∧
(∀ a ∈ u-owns. m_x a = m a)) ∧

(∀ x ≤ l. ∀ ts_x S_x m_x ts'_x S'_x m'_x. c (n + x) = (ts_x,m_x,S_x) → c' (n + x) =
(ts'_x,m'_x,S'_x) →

(∀ j < length ts_x. j = i → ts_x[j] = ts_i) ∧
(∀ a. a ∉ u-owns → a ∉ owned (ts[i]) → S'_x a = S_x a) ∧
(∀ a. a ∉ u-owns → m'_x a = m_x a))

using steps unchanged safe-orig

proof (induct k)
case 0
  show ?case
  apply (rule-tac x=λ l. (u-ts, u-m, u-shared) in exl)
  apply (rule-tac x=0 in exl)
  thm c-n
  apply (simp add: c-n)
  apply (clarsimp simp only: length less_Suc_eq_Suc_excl dest: safe-orig)
done
next
case (Suc k)
  then obtain
trace-suc: trace \( c \cdot n \cdot (Suc \ k) \) and
unchanged-suc: \( \forall l \leq Suc \ k. \forall ts_1 S_1 m_1. c \cdot (n + l) = (ts_1, m_1, S_1) \rightarrow ts_1!i = ts!i \) and
safe-orig: \( \forall x<k. \) safe-delayed \( (c \cdot (n + x)) \)
by simp

interpret direct-computation:
computation direct-memop-step empty-storebuffer-step program-step \( \lambda p \ p' \) is sb. sb.
from trace-suc obtain
trace-c: trace \( c \cdot n \cdot k \) and
last-step: \( c \cdot (n + k) \Rightarrow d \cdot c \cdot (n + (Suc \ k)) \) by (clarsimp simp add: program-trace-def)
from unchanged-suc obtain
unchanged-k: \( \forall l \leq k. \forall ts_1 S_1 m_1. c \cdot (n + l) = (ts_1, m_1, S_1) \rightarrow ts_1!i = ts!i \) and
unchanged-suc-k: \( \forall ts_1 S_1 m_1. c \cdot (n + (Suc \ k)) = (ts_1, m_1, S_1) \rightarrow ts_1!i = ts!i \)
apply apply (rule that)
apply auto
apply (drule-tac x=l in spec)
apply simp
done

from Suc.hyps [OF trace-k unchanged-k safe-orig] obtain \( c' \cdot k \) where
l-k: \( l \leq k \) and
trace-c': trace \( c' \cdot n \cdot l \) and
safe-l: \( \forall x<k. \) safe-delayed \( (c' \cdot (n + x)) \) and
unsafe-l: \( l<k \rightarrow \neg \) safe-delayed \( (c' \cdot (n + l)) \) and
\( c'\cdot n: c' \cdot n = (u-ts, u-m, u-shared) \) and
leq-l: \( \forall x \leq l. \) length \( \text{fst} (c' \cdot (n + x)) \) = length \( \text{fst} (c \cdot (n + x)) \) and
unchanged-i: \( \forall x \leq l. \forall ts_1 S_1 m_1. c \cdot (n + x) = (ts_1, m_1, S_1) \rightarrow c' \cdot (n + x) = (ts_1', m_1', S_1') \rightarrow ts_1'!i = u-ts!i \wedge\)
(\( \forall a \in u-owns. S_1'a = u-shared \ a \)) \wedge
(\( \forall a \in u-owns. S_1' a = S_1 a \)) \wedge
(\( \forall a \in u-owns. m_1'a = u-m a \)) \wedge
(\( \forall a \in u-owns. m_1 a = m_1 a \)) and
sim: \( \forall x \leq l. \forall ts_1 S_1 m_1 ts_1'S_1'm_1'. c \cdot (n + x) = (ts_1, m_1, S_1) \rightarrow c' \cdot (n + x) = (ts_1', m_1', S_1') \rightarrow \)
(\( \forall j<\text{length} ts_1. j \neq i \rightarrow ts_1'!j = ts_1!j \)) \wedge
(\( \forall a. \ a \notin u-owns \rightarrow a \notin \text{owned} (ts_1!) \rightarrow S_1'a = S_1 a \)) \wedge
(\( \forall a. \ a \notin u-owns \rightarrow m_1'a = m_1 a \))
by auto
show ?case
proof (cases \( l<k \))
case True
with True trace-c'\cdot l safe-l unsafe-l unchanged-i sim leq-l c'\cdot n
show ?thesis
apply 
apply (rule-tac x=c' in exI)
apply (rule-tac x=l in exI)
apply auto
done

next

case False
with l-k have l-k: l=k by auto
show ?thesis
proof (cases safe-delayed (c' (n + k)))
case False
with False l-k trace-c'-l safe-l unsafe-l unchanged-i sim leq-l c'-n
show ?thesis
  apply 
  apply (rule-tac x=c' in exI)
  apply (rule-tac x=k in exI)
  apply auto
done

next

case True
note safe-k = this

obtain ts_k S_k m_k where c-k: c (n + k) = (ts_k,m_k,S_k)
  by (cases c (n + k))

obtain ts'_k S'_k m'_k where c-suc-k: c (n + (Suc k)) = (ts'_k,m'_k,S'_k)
  by (cases c (n + (Suc k)))

obtain u-ts_k u-shared_k u-m_k where c'-k: c' (n + k) = (u-ts_k, u-m_k, u-shared_k)
  by (cases c' (n + k))

from trace-preserves-length-ts [OF trace-k, of k 0] c-n c-k i-bound
have i-bound-k: i < length ts_k
  by simp

from leq-l [rule-format, simplified l-k, of k] c-k c'-k
have leq: length u-ts_k = length ts_k
  by simp

note last-step = last-step [simplified c-k c-suc-k]
from unchanged-suc-k c-suc-k
have ts_k !i = ts!i
  by auto
moreover from unchanged-k [rule-format, of k] c-k
have unch-k-i: ts_k !i=ts!i
  by auto
ultimately have ts-eq: ts_k !i=ts_k !i
  by simp

from unchanged-i [simplified l-k, rule-format, OF - c-k c'-k]
obtain
u-ts-eq: u-ts ! i = u-ts ! i and
unchanged-shared: ∀ a ∈ u-owns. u-shared_k a = u-shared a and
unchanged-shared-orig: ∀ a ∈ u-owns. S_k a = S a and
unchanged-owns: ∀ a ∈ u-owns. u-m_k a = u-m a and
unchanged-owns-orig: ∀ a ∈ u-owns. m_k a = m a
by fastforce

from u-ts-eq u-ts-i
have u-ts_k-i: u-ts_k!i = (u-p, u-is, u-tmps, u-sb, u-dirty, u-owns, u-rels)
  by auto
from sim [simplified l-k, rule-format, of k, OF - c-k c′-k]
obtain
ts-sim: (∀ j < length ts_k. j ≠ i → u-ts_k ! j = ts_k ! j) and
shared-sim: (∀ a. a /∈ u-owns → a /∈ owned (ts_k!i) → u-shared_k a = S_k a) and
mem-sim: (∀ a. a /∈ u-owns → u-m_k a = m_k a)
by (auto simp add: unch-k-i)

from unchanged-owns-orig unchanged-owns u-m-shared unchanged-shared
have unchanged-owns-shared: ∀ a. a ∈ u-owns → a ∈ dom u-shared_k → u-m_k a
  = m_k a
  by (auto simp add: simp add: domIff)

from safe-l l-k safe-k
have safe-up-k: ∀ x < k. safe-delayed (c′ (n + x))
  apply clarsimp
  done
  from trace-preserves-simple-ownership-distinct [OF - trace-c′-l] [simplified l-k]
safe-up-k,
    simplified c′-n, simplified, OF dist, of k] c′-k
have dist′: simple-ownership-distinct u-ts_k
  by simp

from trace-preserves-simple-ownership-distinct [OF - trace-k, simplified c-n, simplified, OF dist-ts safe-orig, of k] c-k
have dist-orig′: simple-ownership-distinct ts_k
  by simp

from undo-local-step [OF last-step i-bound-k ts-eq safe-k [simplified c′-k] leq ts-sim u-ts_k-i mem-sim
unchanged-owns-shared shared-sim dist′ dist-orig′]
obtain u-ts′ u-shared′ u-m′ where
  step′: (u-ts_k, u-m_k, u-shared_k) ⇒_d (u-ts′, u-m′, u-shared′) and
ts-eq′: u-ts′ ! i = u-ts_k ! i and
unchanged-shared′: (∀ a ∈ u-owns. u-shared′ a = u-shared_k a) and
unchanged-shared-orig′: (∀ a ∈ u-owns. S_k′ a = S_k a) and
unchanged-owns': (\forall a \in u-owns. u-m' a = u-m_k a) \textbf{and} 
unchanged-owns-orig': (\forall a \in u-owns. m_k' a = m_k a) \textbf{and} 
sim-ts': (\forall j < \text{length ts}'_k. j \neq i \rightarrow u-ts'! j = ts_k'! j) \textbf{and} 
sim-shared': (\forall a. a \notin u-owns \rightarrow a \notin owned (ts_k! i) \rightarrow u-shared' a = S_k' a) \textbf{and} 
sim-m': (\forall a. a \notin u-owns \rightarrow u-m' a = m_k' a) 
by auto

\textbf{define} c'' \textbf{where} c'' == \lambda l. if l \leq n + k then c' l else (u-ts', u-m', u-shared')
\textbf{have} [simp]: \forall x \leq n + k. c'' x = c' x
  by (auto simp add: c''-def)
\textbf{have} [simp]: c'' (Suc (n + k)) = (u-ts', u-m', u-shared')
  by (auto simp add: c''-def)

from trace-c' l-k step' c' k \textbf{have} trace': trace c'' n (Suc k)
  apply (simp add: program-trace-def)
  apply (clarsimp simp add: split-less-Suc)
  done

from direct-computation.step-preserves-length-ts [OF last-step]
  have leq-ts_k': \text{length ts}'_k = \text{length ts}_k.

with direct-computation.step-preserves-length-ts [OF step'] leq
  have leq': \text{length u-ts'} = \text{length ts}_k
    by simp
  show \textbf{thesis}
    apply (rule-tac x=c'' in exI)
    apply (rule-tac x=Suc k in exI)
    using safe-l l-k unch-k-i sim-c-suc-k leq-l c'-n leq'
    apply (clarsimp simp add: split-less-Suc split-le-Suc safe-k trace' leq-ts_k' sim-ts'
    sim-shared' sim-m' unch-k-i
      ts-eq' u-ts-eq
      unch-shared' unch-shared unch-shared-orig
      unch-shared-orig'
      unch-owns' unch-owns
      unch-owns-orig' unch-owns-orig )
  done
  qed
qed

locale program-safe-reach-upto = program +
  fixes n fixes safe fixes c_0
  assumes safe-config: \[ k \leq n; trace c 0 k; c 0 = c_0; 1 \leq k \] \rightarrow safe (c 1)

abbreviation (in program)
  safe-reach-upto \equiv program-safe-reach-upto program-step

lemma (in program) safe-reach-upto-le:
assumes safe: safe-reach-upto n safe c
assumes m-n: m ≤ n
shows safe-reach-upto m safe c
using safe m-n
apply (clarsimp simp add: program-safe-reach-upto-def)
subgoal for k c
  apply (subgoal-tac k ≤ n)
  apply blast
  apply simp
done
done

lemma (in program) last-action-of-thread:
assumes trace: trace c 0 k
shows
— thread i never executes
(∀1 ≤ k. fst (c l)!i = fst (c k)!i) ∨
— thread i has a last step in the trace
(∃last < k. fst (c last)!i ≠fst (c (Suc last))!i ∧
(∀l. last < 1 → l ≤ k → fst (c l)!i = fst (c k)!i))
using trace
proof (induct k)
case 0 thus ?case
  by auto
next
case (Suc k)
hence trace c 0 (Suc k) by simp
then obtain
  trace-k: trace c 0 k and
  last-step: c k ⇒ₐ c (Suc k)
  by (clarsimp simp add: program-trace-def)
show ?case
proof (cases fst (c k)!i = fst (c (Suc k))!i)
case False
then show ?thesis
  apply —
  apply (rule disjI2)
  apply (rule-tac x=k in exI)
  apply clarsimp
  apply (subgoal-tac l=Suc k)
  apply auto
  done
next
case True
note idle-i = this
\{ 
assume same: (\forall l \leq k. \text{fst (c l)} ! i = \text{fst (c k)} ! i) 

have ?thesis 
  apply – 
  apply (rule disjI1) 
  apply clarsimp 
  apply (case-tac l=Suc k) 
  apply (simp add: idle-i) 
  apply (rule same [simplified idle-i, rule-format]) 
  apply simp 
  done 
\}

moreover 
{ 
fix last 
assume last-k: last < k 
assume last-step: \text{fst (c last)} ! i \neq \text{fst (c (Suc last))} ! i 
assume idle: (\forall l > last. l \leq k \rightarrow \text{fst (c l)} ! i = \text{fst (c k)} ! i) 
have ?thesis 
  apply – 
  apply (rule disjI2) 
  apply (rule-tac x=last in exI) 
  using last-k 
  apply (simp add: last-step) 
  using idle [simplified idle-i] 
  apply clarsimp 
  apply clarsimp 
  apply clarsimp 
  done 
}\}%

moreover note Suc.hyps [OF trace-k] 
ultimately 
show ?thesis 
  by blast 
qed 
qed

lemma (in program) sequence-traces: 
assumes trace1: trace c_1 0 k 
assumes trace2: trace c_2 m l 
assumes seq: c_2 m = c_1 k 
assumes c-def: c = (\lambda x. if x \leq k then c_1 x else (c_2 (m + x - k))) 
shows trace c 0 (k + l) 
proof – 
  from trace1 
  interpret trace1: program-trace program-step c_1 0 k . 
  from trace2 
  interpret trace2: program-trace program-step c_2 m l .
{ 
  fix x
  assume x-bound: x < (k + l)
  have c x ⇒ₜ c (Suc x)
  proof (cases x < k)
    case True
    from trace1.step [OF True] True
    show ?thesis
    by (simp add: c-def)
  next
    case False
    hence k-x: k ≤ x
    by auto
    with x-bound have bound: x − k < l
    by auto
    from k-x have eq: (Suc (m + x) − k) = Suc (m + x − k)
    by simp
    from trace2.step [OF bound] k-x seq
    show ?thesis
    by (auto simp add: c-def eq)
  qed
}
thus ?thesis
by (auto simp add: program-trace-def)
qed

theorem (in program-progress) safe-free-flowing-implies-safe-delayed:
  assumes init: initial c₀
  assumes dist: simple-ownership-distinct (fst c₀)
  assumes read-only-unowned: read-only-unowned (snd (snd c₀)) (fst c₀)
  assumes unowned-shared: unowned-shared (snd (snd c₀)) (fst c₀)
  assumes safe-reach-ff: safe-reach-upto n safe-free-flowing c₀
  shows safe-reach-upto n safe-delayed c₀
using safe-reach-ff
proof (induct n)
  case 0
  hence safe-reach-upto 0 safe-free-flowing c₀ by simp
  hence safe-free-flowing c₀
  by (auto simp add: program-safe-reach-upto-def)
  from initial-safe-free-flowing-implies-safe-delayed [OF init this]
  have safe-delayed c₀.
  then show ?case
  by (simp add: program-safe-reach-upto-def)
next
  case (Suc n)
  hence safe-reach-suc: safe-reach-upto (Suc n) safe-free-flowing c₀ by simp
  then interpret safe-reach-suc-inter: program-safe-reach-upto program-step (Suc n)
  safe-free-flowing c₀ .
  from safe-reach-upto-le [OF safe-reach-suc ]
  have safe-reach-n: safe-reach-upto n safe-free-flowing c₀ by simp

88
from Suc.hyps [OF this]
have safe-delayed-reach-n: safe-reach-upto n safe-delayed c_0.
then interpret safe-delayed-reach-inter: program-safe-reach-upto program-step n safe-delayed c_0.
interpret direct-computation:
  computation direct-memop-step empty-storebuffer-step program-step λp p' is sb. sb.
show ?case
proof (cases safe-reach-upto (Suc n) safe-delayed c_0)
  case True thus ?thesis.
next
case False
from safe-delayed-reach-n False
obtain c where
  trace: trace c 0 (Suc n) and
c-0: c 0 = c_0 and
safe-delayed-upto-n: ∀k≤n. safe-delayed (c k) and
violation-delayed-suc: ¬ safe-delayed (c (Suc n))
proof
  from False obtain k l where
    k-suc: k ≤ Suc n and
    trace-k: trace c 0 k and
    l-k: l ≤ k and
    violation: ¬ safe-delayed (c l) and
    start: c 0 = c_0
    by (clarsimp simp add: program-safe-reach-upto-def)
show ?thesis
proof (cases k = Suc n)
  case False
  with k-suc have k ≤ n
  by auto
  from safe-delayed-reach-inter.safe-config [where c=c, OF this trace-k start l-k]
  have safe-delayed (c l),
  with violation have False by simp
  thus ?thesis ..
next
case True
note k-suc-n = this
from trace-k True have trace-n: trace c 0 n
  by (auto simp add: program-trace-def)
show ?thesis
proof (cases l=Suc n)
  case False
  with k-suc-n l-k have l ≤ n by simp
  from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n start this ]
  have safe-delayed (c l) by simp
  with violation have False by simp
  thus ?thesis ..
next
case True
from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n start]
have \( \forall k \leq n. \) safe-delayed \((c \, k)\) by simp
with True k-suc-n trace-k start violation
show \(?thesis
  apply –
  apply (rule that)
  apply auto
done
qed
qed
qed

from trace_interpret trace-inter: program-trace program-step c 0 Suc n .

from safe-reach-suc-inter.safe-config [where c=c, OF - trace-c-0]
have safe-suc: safe-free-flowing \((c \, (Suc \, n))\)
  by auto

obtain ts S m where c-suc: c \((Suc \, n)\) = \((ts,\,m,S)\) by \((cases c \, (Suc \, n))\)
from violation-delayed-suc c-suc
obtain i p is \(\emptyset\) sb D O R where
  i-bound: \(i < \text{length ts}\) and
  ts-i: \(\text{ts}!\,i = (p,\text{is},\emptyset,\text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R})\) and
  violation-i: \(\neg \text{map owned ts},\text{map released ts},i \vdash (\text{is},\emptyset,\text{m},\mathcal{D},\mathcal{O},\mathcal{S})\)√
  by \((\text{fastforce simp add: safe-free-flowing-def safe-delayed-def})\)

from trace-preserves-unowned-shared [where c=c and n=0 and l=Suc n,
simplified c-0, OF dist unowned-shared trace] safe-delayed-upto-n c-suc
have unowned-shared \(S\) ts by auto
then interpret unowned-shared \(S\) ts .

from violation-i obtain ins is’ where is: is = ins#is’
  by \((cases is)\) \((auto simp add: safe-delayed-direct-memop-state.Nil)\)
from safeE [OF safe-suc [simplified c-suc] i-bound ts-i]
have safe-i: map owned ts,i \vdash (is,\emptyset,\text{m},\mathcal{D},\mathcal{O},S)\)√.

define races where races == \(\lambda \mathcal{R}.\) (case ins of
  Read volatile a t \(\Rightarrow (\mathcal{R} \, a = \text{Some False}) \lor (\neg \text{volatile} \land a \in \text{dom } \mathcal{R})\)
  | Write volatile a sop A L R W \(\Rightarrow (a \in \text{dom } \mathcal{R} \lor (\text{volatile} \land A \cap \text{dom } \mathcal{R} \neq \{}))\)
  | Ghost A L R W \(\Rightarrow (A \cap \text{dom } \mathcal{R} \neq \{}))\)
  | RMW a t (D,f) cond ret A L R W \(\Rightarrow (\text{if cond } (\emptyset(t \mapsto m \, a))\)
  | then a \(\in \text{dom } \mathcal{R} \lor A \cap \text{dom } \mathcal{R} \neq \{}\)
  | else \(\mathcal{R} \, a = \text{Some False})\)
  | - \(\Rightarrow \text{False}\)\)

90
\{ 
assume no-race: \\
\forall j. j < \text{length} ts \rightarrow j \neq i \rightarrow \neg \text{races} (\text{released} (ts!j)) 
from safe-i 
have map owned ts,map released ts,i \vdash (is,0,m,D,O,S)\checkmark 
proof cases 
  case Read 
    thus \thesis 
    using is no-race 
    by (auto simp add: races-def intro: safe-delayed-direct-memop-state.intros) 
next 
  case WriteNonVolatile 
    thus \thesis 
    using is no-race 
    by (auto simp add: races-def intro: safe-delayed-direct-memop-state.intros) 
next 
  case WriteVolatile 
    thus \thesis 
    using is no-race 
    apply (clarsimp simp add: races-def) 
    apply (rule safe-delayed-direct-memop-state.intros) 
    apply auto 
    done 
next 
  case Fence 
    thus \thesis 
    using is no-race 
    by (auto simp add: races-def intro: safe-delayed-direct-memop-state.intros) 
next 
  case Ghost 
    thus \thesis 
    using is no-race 
    apply (clarsimp simp add: races-def) 
    apply (rule safe-delayed-direct-memop-state.intros) 
    apply auto 
    done 
next 
  case RMWReadOnly 
    thus \thesis 
    using is no-race 
    by (auto simp add: races-def intro: safe-delayed-direct-memop-state.intros) 
next 
  case (RMWWrite cond t a - - A - O) 
    thus \thesis 
    using is no-race unowned-shared’ [rule-format, of a] ts-i 
    apply (clarsimp simp add: races-def) 
    apply (rule safe-delayed-direct-memop-state.RMWWrite) 
    apply auto 
    apply force 
    done 
\}
next
  case Nil with is show ?thesis by auto
qed
}
with violation-i
obtain j where
  j-bound: j < length ts and
  neq-j-i: j ≠ i and
  race: races (released (ts!j))
by auto

obtain p_j is_j v_j sb_j D_j O_j R_j where
  ts-j: ts!j = (p_j, is_j, v_j, sb_j, D_j, O_j, R_j)
apply (cases ts!j)
apply force
done

from race
have R_j-non-empty: R_j ≠ Map.empty
  by (auto simp add: ts-j races-def split: instr.splits if-split-asm)

{ assume idle-j: ∀l≤Suc n. fst (c l) ! j = fst (c (Suc n)) ! j
  have ?thesis
  proof –
    from idle-j [rule-format, of 0] c-suc c-0 ts-j
    have c_0-j: fst c_0 ! j = ts!j
      by clarsimp
    from trace-preserves-length-ts [OF trace, of 0 Suc n] c-0 c-suc
    have length (fst c_0) = length ts
      by clarsimp
    with j-bound have j < length (fst c_0)
      by simp
    with nth-mem [OF this] init c_0-j ts-j
    have R_j = Map.empty
      by (auto simp add: initial-def)
    with R_j-non-empty have False
      by simp
    thus ?thesis ..
  qed
}
moreover
{ fix last
  assume last-bound: last<Suc n
  assume last-step-changed-j: fst (c last) ! j ≠ fst (c (Suc last)) ! j
  assume idle-rest: ∀l>last. l ≤ Suc n → fst (c l) ! j = fst (c (Suc n)) ! j
  have ?thesis
  proof –
    obtain ts_l S_l m_l where
c-last: c last = (ts, ml, Sl) 

by (cases c last)

obtain ts′, S′ where 
  c-last': c (Suc last) = (ts′, ml′, S′) 
  by (cases c (Suc last))

from idle-rest [rule-format, of Suc last] c-suc c-last’ last-bound
have ts′: ts′[j] = ts[j]
  by auto

from last-step-changed-j c-last c-last’
have j-changed: ts′[j] ≠ ts[j]
  by auto

from trace-inter-step [OF last-bound] c-last c-last’
have last-step: (ts, ml, Sl) ⇒ₜ (ts′, ml′, S′)
  by simp

obtain p l is l θ l sb l D l O l R l where 
  ts′: ts′[j] = (p l, is l, θ l, sb l, D l, O l, R l)
  apply (cases ts′[j])
  apply force
  done

from trace-preserves-length-ts [OF trace, of last Suc n] c-last c-suc last-bound
have leq: length ts = length ts
  by simp

with j-bound have j-bound: j < length ts
  by simp

from trace have trace-n: trace c 0 n
  by (auto simp add: program-trace-def)

  from safe-delayed-reach-inter.safe-config [where k=n and c=c and l=last, OF - 
  trace-n c 0] last-bound c-last
have safe-delayed-last: safe-delayed (ts, ml, Sl)
  by auto

from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n c 0]
have safe-delayed-upto-n: ∀ x<n. safe-delayed (c (0 + x))
  by auto

from trace-preserves-simple-ownership-distinct [where c=c and n=0 and l=last, 
  simplifed c 0, OF dist trace-n safe-delayed-upto-n]
last-bound c-last
have dist-last: simple-ownership-distinct ts
  by auto

from trace-preserves-read-only-unowned [where c=c and n=0 and l=last, 
  simplifed c 0, OF dist read-only-unowned trace-n safe-delayed-upto-n]
last-bound c-last
have ro-last-last: read-only-unowned S₁ ts
by auto

from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n c-0]
have safe-delayed-upto-suc-n: \( \forall x < \text{Suc } n. \text{ safe-delayed } (c (0 + x)) \)
  by auto

from trace-preserves-simple-ownership-distinct [where c=c and n=0 and l=Suc last,
  simplified c-0, OF dist trace safe-delayed-upto-suc
last-bound c-last'
have dist-last': simple-ownership-distinct ts'l'
  by auto
from trace last-bound have trace-last: trace c 0 last
  by (auto simp add: program-trace-def)
from trace last-bound have trace-rest: trace c (Suc last) (n - last)
  by (auto simp add: program-trace-def)
from idle-rest last-bound
have idle-rest':
  \[ \forall l \leq n - \text{last}.
      \forall ts'_l. S'_l, m'_l, c (\text{Suc last } + l) = (ts_l, m_l, S_l) \rightarrow ts'_l ! j = ts_l' ! j \]
  apply clarsimp
  apply (drule-tac x=Suc (last + l) in spec)
  apply (auto simp add: c-last' c-suc ts_l'j)
  done
from safe-delayed-upto-suc-n [rule-format, of last] last-bound
have safe-delayed-last: safe-delayed (ts_l, m_l, S_l)
  by (auto simp add: c-last)
from safe-delayedE [OF this j-bound ts_l-j] have safeq: map owned ts_l,map released ts_l j-(is_l, \theta_l, m_l, D_l, O_l, S_l)√.
from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n c-0]
have safe-delayed-upto-last: \( \forall x < n - \text{last}. \text{ safe-delayed } (c (\text{Suc } (\text{last } + x))) \)
  by auto
from last-step
show ?thesis
proof (cases)
  case (Program i' - - - - - - - - p'is')
  with j-changed j-bound'{ts_l-j}
  obtain
    ts_l': ts_l' = ts_l[j:==(p',is_l@is',\theta_l,sb_l,D_l,O_l,R_l)] and
    S_l': S_l'=S_l and
    m_l': m_l'=m_l and
    prog-step: \( \theta_l \vdash p_l \rightarrow p' (p', is') \)
  by (cases i'=j) auto
from \( ts_l \cdot j \) ts_j \cdot j \cdot j-bound_i

obtain eqs: \( p' = p_j \) is@is' = is_j \( \theta_l = \theta_j \) \( D_l = D_j \) \( O_l = O_j \) \( R_l = R_j \)

by auto

from undo-local-steps [where \( c = c \), OF trace-rest c-last' idle-rest' safe-delayed-last, simplified ts', simplified, OF j-bound_i ts_j \cdot j \cdot j-bound]

obtain c' \( k \) where

k-bound: \( k \leq n - \) last and

trace-c': trace c' (Suc last) \( k \) and

c'-first: c' (Suc last) = (ts_l, m_l, S_l) and

c'-leq: (\( \forall x \leq k \). length (fst (c' (Suc (last + x)))) = length (fst (c (Suc (last + x)))))) and

c'-safe: (\( \forall x < k \). safe-delayed (c' (Suc (last + x)))) and

c'-unsafe: (k < n - last \( \rightarrow \) ¬ safe-delayed (c' (Suc (last + k)))) and

c'-unch:

(\( \forall x \leq k \). \( \forall ts_x S_x m_x \).

c (Suc (last + x)) = (ts_x, m_x, S_x) \( \rightarrow \)

(\( \forall ts'_x S'_x m'_x \).

c' (Suc (last + x)) = (ts'_x, m'_x, S'_x) \( \rightarrow \)

ts'_x ! j = ts_l ! j \( \land \)

(\( \forall a \in O_l \). S'_x a = S_l a) \( \land \)

(\( \forall a \in O_l \). S'_x a = S_l a) \( \land \)

(\( \forall a \in O_l \). m'_x a = m_l a) \( \land \) (\( \forall a \in O_l \). m'_x a = m_l a)) and

and

and

by auto

obtain c Undo where c-undo: c-undo = (\( \lambda x \). if x \( \leq \) last then c x else c' (Suc last + x - last))

by blast

have c-undo-0: c-undo 0 = c_0

by (auto simp add: c-undo c-0)

from sequence-traces [OF trace-last trace-c', simplified c-last, OF c'-first c-undo]

have trace-undo: trace c-undo 0 (last + k).

obtain u-ts u-shared u-m where

c-undo-n: c-undo n = (u-ts, u-m, u-shared)

by (cases c-undo n)

with last-bound c'-first c-last

have c'-suc: c' (Suc n) = (u-ts, u-m, u-shared)

apply (auto simp add: c-undo split: if-split-asm)
apply (subgoal-tac n=last)
apply auto
done

show ?thesis
proof (cases k < n - last)
case True
  with c'-unsafe have unsafe: ¬ safe-delayed (c-undo (last + k))
  by (auto simp add: c-undo c-last c'-first)
from True have last + k ≤ n
  by auto
from safe-delayed-reach-inter.safe-config [OF this trace-undo, of last + k]
have safe-delayed (c-undo (last + k))
  by (auto simp add: c-undo c-0)
with unsafe have False by simp
thus ?thesis ..
next
case False
  with k-bound have k: k = n - last
  by auto
have eq': Suc (last + (n - last)) = Suc n
  using last-bound
  by simp
from c'-unch [rule-format, of k, simplified k eq', OF - c-suc c'-suc]
obtain u-ts-j: u-ts!j = ts!j and
  shared-unch: ∀a∈O. u-shared a = S_1 a and
  shared-orig-unch: ∀a∈O. S a = S_1 a and
  mem-unch: ∀a∈O. u-m a = m_l a and
  mem-unch-orig: ∀a∈O. m a = m_l a
  by auto
from c'-sim [rule-format, of k, simplified k eq', OF - c-suc c'-suc] i-bound neq-j-i
obtain u-ts-i: u-ts!i = ts!i and
  shared-sim: ∀a. a /∈ O_l → u-shared a = S a and
  mem-sim: ∀a. a /∈ O_l → u-m a = m a
  by auto
from c'-leq [rule-format, of k] c'-suc c-suc
have leq-u-ts: length u-ts = length ts
  by (auto simp add: eq' k)
from j-bound leq-u-ts
have j-bound-u: j < length u-ts
  by simp
from i-bound leq-u-ts
have i-bound-u: i < length u-ts
  by simp
from k last-bound have l-k-eq: last + k = n
  by auto

96
from safe-delayed-reach-inter.safe-config [OF - trace-undo, simplified l-k-eq]
  k c-0 last-bound
have safe-delayed-c-undo': \( \forall x \leq n \). safe-delayed (c-undo x)
  by (auto simp add: c-undo split: if-split-asm)
hence safe-delayed-c-undo: \( \forall x < n \). safe-delayed (c-undo x)
  by (auto)
from trace-preserves-simple-ownership-distinct [OF - trace-undo,
  simplified l-k-eq c-undo-0, simplified, OF dist this, of n] dist c-undo-n
have dist-u-ts: simple-ownership-distinct u-ts
  by auto
then interpret dist-u-ts-inter: simple-ownership-distinct u-ts .

{"fix a
  have u-m a = m a
proof (cases a \in O_l)
  case True with mem-unch
  have u-m a = m_l a
    by auto
  moreover
  from True mem-unch-orig
  have m a = m_l a
    by auto
  ultimately show ?thesis by simp
next
  case False
  with mem-sim
  show ?thesis
    by auto
qed
} hence u-m-eq: u-m = m by (rule ext, auto)

{"fix a
  have u-shared a = S a
proof (cases a \in O_l)
  case True with shared-unch
  have u-shared a = S_l a
    by auto
  moreover
  from True shared-orig-unch
  have S a = S_l a
    by auto
  ultimately show ?thesis by simp
next
  case False
  with shared-sim
  show ?thesis
    by auto
qed

97
\{ \textbf{hence} \textit{u-shared-Eq: } u\textit{-shared} = S \textit{ by } \textit{ (rule ext, auto)} \\

\{ \\
\textbf{assume} \textit{safe: map owned u-ts, map released u-ts, i \rightarrow (is, o, u-m, D, O, u\text{-shared})} \\
\textbf{then have} \textit{False} \\
\textbf{proof} \textbf{cases} \\
\textbf{case} \textit{Read} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{WriteNonVolatile} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{WriteVolatile} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{apply} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{apply} \textit{fastforce} \\
\textbf{done} \\
\textbf{next} \\
\textbf{case} \textit{Fence} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{Ghost} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{apply} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{apply} \textit{fastforce} \\
\textbf{done} \\
\textbf{next} \\
\textbf{case} \textit{(RMWReadOnly cond t a D f ret A L W)} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{RMWWrite} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{apply} \textit{(auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm)} \\
\textbf{apply} \textit{fastforce+} \\
\textbf{done} \\
\textbf{next} \\
\textbf{case} \textit{Nil} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{proof} \textbf{cases} \\
\textbf{case} \textit{Read} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{WriteNonVolatile} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{WriteVolatile} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{apply} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{apply} \textit{fastforce} \\
\textbf{done} \\
\textbf{next} \\
\textbf{case} \textit{Fence} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{Ghost} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{apply} \textit{(auto simp add:eqs races-def split: if-split-asm)} \\
\textbf{apply} \textit{fastforce} \\
\textbf{done} \\
\textbf{next} \\
\textbf{case} \textit{(RMWReadOnly cond t a D f ret A L W)} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm)} \\
\textbf{next} \\
\textbf{case} \textit{RMWWrite} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{apply} \textit{(auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm)} \\
\textbf{apply} \textit{fastforce+} \\
\textbf{done} \\
\textbf{next} \\
\textbf{case} \textit{Nil} \\
\textbf{then show} \textit{?thesis} \\
\textbf{using} \textit{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \\
\textbf{by} \textit{(auto simp add:eqs races-def split: if-split-asm)}
by (auto simp add:eqs races-def split: if-split-asm)

qed

hence \neg\text{safe-delayed (u-ts, u-m, u-shared)}

apply (clarsimp simp add: safe-delayed-def)

apply (rule-tac x=i in exI)

using u-ts-i ts-i i-bound-u

apply auto

done

moreover

from safe-delayed-c-undo' [rule-format, of n] c-undo-n

have safe-delayed (u-ts, u-m, u-shared)

by simp

ultimately have False

by simp

thus \?thesis

by simp

qed

next

case (Memop i' - - - - - - - is l' θ l' sb l' D l' O l' R l' )

with j-changed j-bound

obtain

ts_1': ts_1' = ts_1[j:]=(p_i,is_i',i'_1, sb_1', D_1', O_1', R_1') and

mem-step: (is_i, i_1, sb_1, m_1, D_1, O_1, R_1, S_1) \rightarrow

(is_i', i'_1, sb_1', m_1', D_1', O_1', R_1', S_1')

by (cases i'=j) auto

from mem-step

show \?thesis

proof (cases)

case (Read volatile a t)

then obtain

is: is_i = Read volatile a t \# is_i' and

i_1': i_1' = i_1(t \mapsto m_1 a) and

sb_1': sb_1'=sb_1 and

D_1': D_1'=D_1 and

O_1': O_1'= O_1 and

R_1': R_1'= R_1 and

S_1': S_1'=S_1 and

m_1': m_1'= m_1

by auto

note eqs' = \theta_1' sb_1' D_1' O_1' R_1' S_1' m_1'

from ts_1'-j ts_1' ts-j j-bound_i eqs'

obtain eqs: p_i=p_j is_i'=is_j \theta_i(t \mapsto m_i a)=\theta_j D_i=D_j O_i=O_j R_i=R_j

by auto

from undo-local-steps [where c=c, OF trace-rest c-last idle-rest safe-delayed-last, simplified ts_1',

simplified,
OF j-bound ts-j [simplified], simplified ml′ S′j, simplified, OF dist-last dist-last′ [simplified ts′j, simplified] safe-delayed-upto-last

**obtain c′ k where**

k-bound: k ≤ n − last and

trace-c′: trace c′ (Suc last) k and
c′-first: c′ (Suc last) = (tsl, ml, S1) and
c′-leq: (∀x≤k. length (fst (c′ (Suc (last + x)))) = length (fst (c (Suc (last + x))))) and

c′-safe: (∀x<k. safe-delayed (c′ (Suc (last + x)))) and
c′-unsafe: (k < n − last −→ ¬ safe-delayed (c′ (Suc (last + k)))) and
c′-unch:

(∀x≤k. ∀tsx Sx mx.
  c (Suc (last + x)) = (tsx, mx, Sx) −→
  (∀tsx′ Sx′ mx′.
    c′ (Suc (last + x)) = (tsx′, mx′, Sx′) −→
    tsx′! j = tsl! j ∧
    (∀a∈Ol. Sx′ a = S1 a) ∧
    (∀a∈Ol. Sx a = S1 a) ∧
    (∀a∈Ol. mx′ a = ml a) ∧ (∀a∈Ol. mx a = ml a))) and

c′-sim:

(∀x≤k. ∀tsx Sx mx.
  c (Suc (last + x)) = (tsx, mx, Sx) −→
  (∀tsx′ Sx′ mx′.
    c′ (Suc (last + x)) = (tsx′, mx′, Sx′) −→
    (∀ja<length tsx. ja ≠ j −→ tsx′! ja = tsx! ja) ∧
    (∀a. a /∈ Ol −→ Sx′ a = Sx a) ∧
    (∀a. a /∈ Ol −→ mx′ a = mx a)))

by (clarsimp simp add: Ol′)

**obtain c-undo where** c-undo: c-undo = (λx. if x ≤ last then c x else c′ (Suc last + x − last))

by blast

**have** c-undo-0: c-undo 0 = c0

by (auto simp add: c-undo c0)

from sequence-traces [OF trace-last trace-c′, simplified c-last, OF c′-first c-undo]

**have** trace-undo: trace c-undo 0 (last + k).

**obtain** u-ts u-shared u-m where

  c-undo-n: c-undo n = (u-ts, u-m, u-shared)

by (cases c-undo n)

**with** last-bound c′-first c-last

**have** c′-suc: c′ (Suc n) = (u-ts, u-m, u-shared)

**apply** (auto simp add: c-undo split: if-split_asm)

**apply** (subgoal-tac n=last)

**apply** auto

**done**

show ?thesis

**proof** (cases k < n − last)

**case** True

with c′-unsafe **have** unsafe: ¬ safe-delayed (c-undo (last + k))

100
by (auto simp add: c-undo c-last c'-first)
from True have last + k ≤ n
  by auto
from safe-delayed-reach-inter.safe-config [OF this trace-undo, of last + k]
have safe-delayed (c-undo (last + k))
  by (auto simp add: c-undo c-0)
with unsafe have False by simp
thus ?thesis ..
next
case False
with k-bound have k: k = n - last
  by auto
have eq': Suc (last + (n - last)) = Suc n
  using last-bound
  by simp
from c'-unch [rule-format, of k, simplified k eq', OF - c-suc c'-suc]
obtain u-ts-j: u-ts![j] = ts![j] and
  shared-unch: ∀a∈O, u-shared a = S I a and
  shared-orig-unch: ∀a∈O, S a = S I a and
  mem-unch: ∀a∈O, u-m a = m I a and
  mem-unch-orig: ∀a∈O, m a = m I a
  by auto
from c'-sim [rule-format, of k, simplified k eq', OF - c-suc c'-suc] i-bound neq-j-i
obtain u-ts-i: u-ts![i] = ts![i] and
  shared-sim: ∀a. a /∈ O → u-shared a = S a and
  mem-sim: ∀a. a /∈ O → u-m a = m a
  by auto
from c'-leq [rule-format, of k] c'-suc c-suc
have leq-u-ts: length u-ts = length ts
  by (auto simp add: eq' k)
from j-bound leq-u-ts
have j-bound-u: j < length u-ts
  by simp
from i-bound leq-u-ts
have i-bound-u: i < length u-ts
  by simp
from k last-bound have l-k-eq: last + k = n
  by auto
from safe-delayed-reach-inter.safe-config [OF - trace-undo, simplified l-k-eq]
  k c-0 last-bound
have safe-delayed-c-undo': ∀ x≤n. safe-delayed (c-undo x)
  by (auto simp add: c-undo split: if-split_asm)
hence safe-delayed-c-undo: ∀ x<n. safe-delayed (c-undo x)
  by (auto)
from trace-preserves-simple-ownership-distinct [OF - trace-undo, simplified l-k-eq c-undo-0, simplified, OF dist this, of n] dist c-undo-n
have dist-u-ts: simple-ownership-distinct u-ts
then interpret dist-u-ts-inter: simple-ownership-distinct u-ts.

{ 
  fix a
  have u-m a = m a
  proof (cases a ∈ O_l)
    case True with mem-unch
    have u-m a = m l a
      by auto
    moreover
    from True mem-unch-orig
    have m a = m l a
      by auto
    ultimately show ?thesis by simp
  next
    case False
    with mem-sim
    show ?thesis
      by auto
  qed
} hence u-m-eq: u-m = m − (rule ext, auto)

{ 
  fix a
  have u-shared a = S a
  proof (cases a ∈ O_l)
    case True with shared-unch
    have u-shared a = S l a
      by auto
    moreover
    from True shared-orig-unch
    have S a = S l a
      by auto
    ultimately show ?thesis by simp
  next
    case False
    with shared-sim
    show ?thesis
      by auto
  qed
} hence u-shared-eq: u-shared = S by − (rule ext, auto)

{ 
  assume safe: map owned u-ts, map released u-ts, i ⊢ (is, θ, u-m, D, O, u-shared) √
  then have False
  proof cases
    case Read
    then show ?thesis
using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
by (auto simp add:eqs races-def split: if-split-asm)
next
  case WriteNonVolatile
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  by (auto simp add:eqs races-def split: if-split-asm)
next
  case WriteVolatile
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  apply (auto simp add:eqs races-def split: if-split-asm) 
  apply fastforce
  done
next
  case Fence
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  by (auto simp add:eqs races-def split: if-split-asm)
next
  case Ghost
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  apply (auto simp add:eqs races-def split: if-split-asm) 
  apply fastforce
  done
next
  case (RMWReadOnly cond t a D f ret A L R W)
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  by (auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm)
next
  case RMWWrite
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  apply (auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm) 
  apply fastforce+
  done
next
  case Nil
  then show ?thesis
  using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j 
  by (auto simp add:eqs races-def split: if-split-asm)
qed

hence ¬ safe-delayed (u-ts, u-m, u-shared)
apply (clarsimp simp add: safe-delayed-def) 
apply (rule-tac x=i in exl)
using u-ts-i ts-i i-bound-u 
apply auto

103
done
moreover
from safe-delayed-c-undo [rule-format, of n] c-undo-n
have safe-delayed (u-ts, u-m, u-shared)
  by simp
ultimately have False
  by simp
thus ?thesis
  by simp
qed
next
case (WriteNonVolatile a D f A L R W)
then obtain
  is_l: is_l = Write False a (D, f) A L R W # is_l' and
  t_l': t_l' = t_l and
  s_l': s_l' = s_l and
  D_l': D_l' = D_l and
  O_l': O_l' = O_l and
  R_l': R_l' = R_l and
  S_l': S_l' = S_l and
  m_l': m_l' = m_l(a := f t_l)
  by auto
note eqs' = t_l' s_l' D_l' O_l' R_l' S_l' m_l'
from ts_l'-j ts_l' ts-j-bound_l eqs'
obtain eqs: p_l = p_j is_l' = is_j t_l' = t_j D_l' = D_j O_l' = O_j
  R_l' = R_j
  by auto

from safe [simplified is]
obtain a-owned: a ∈ O_l and a-unshared: a /∈ dom S_l
  by cases auto
have m_l-unch-unowned: ∀ a'. a' /∈ O_l → m_l a' = (m_l(a := f t_l)) a'
  using a-owned by auto
have m_l-unch-unshared: ∀ a'. a' ∈ O_l → a' ∈ dom S_l → m_l a' = (m_l(a := f t_l)) a'
  using a-unshared by auto

from undo-local-steps [where c=c, OF trace-rest c-last' idle-rest' safe-delayed-last, simplified ts_l',
  simplified,
  OF j-bound_l ts_l-j [simplified], simplified m_l' S_l', OF m_l-unch-unowned
  m_l-unch-unshared, simplified,
  OF dist-last dist-last' [simplified ts_l', simplified] safe-delayed-upto-last]
obtain c' k where
  k-bound: k ≤ n - last and
  trace-c': trace c' (Suc last) k and
  c'-first: c' (Suc last) = (ts_l, m_l, S_l) and

104
\(c'\text{-leq}: (\forall x \leq k. \text{length (fst (} c' (\text{Suc (last + x)))}) = \text{length (fst (} c (\text{Suc (last + x))))}) \) and
\(c'\text{-safe}: (\forall x < k. \text{safe-delayed (} c' (\text{Suc (last + x)))}) \) and
\(c'\text{-unsafe}: (k < n - \text{last} \rightarrow \neg \text{safe-delayed (} c' (\text{Suc (last + k)))}) \) and
\(c'\text{-unch}\):

\[
(\forall x \leq k. \forall ts_x S_x m_x.
\begin{align*}
&c (\text{Suc (last + x)}) = (ts_x, m_x, S_x) \rightarrow \nonumber \\
&c' (\text{Suc (last + x)}) = (ts_x', m_x', S_x') \rightarrow \nonumber \\
&ts_x'! j = ts_l! j \wedge \\
&(\forall a \in O_l. S_x' a = S_l a) \wedge \\
&(\forall a \in O_l. S_x a = S_l a) \wedge \\
&(\forall a \in O_l. m_x' a = m_l a) \wedge (\forall a' \in O_l, m_x a' = (m_l(a := f \? l)) a'))
\end{align*}
\]
and
\(c'\text{-sim}\):

\[
(\forall x \leq k. \forall ts_x S_x m_x.
\begin{align*}
&c (\text{Suc (last + x)}) = (ts_x, m_x, S_x) \rightarrow \nonumber \\
&c' (\text{Suc (last + x)}) = (ts_x', m_x', S_x') \rightarrow \nonumber \\
&(\forall a \in O_l. S_x' a = S_l a) \wedge \\
&(\forall a \in O_l. S_x a = S_l a) \wedge \\
&(\forall a \in O_l. m_x' a = m_l a) \wedge (\forall a' \in O_l, m_x a' = (m_l(a := f \? l)) a'))
\end{align*}
\]
by (clarsimp simp add: O_l)

obtain \(c\text{-undo where} c\text{-undo}: c\text{-undo} = (\lambda x. \text{if } x \leq \text{last then } c x \text{ else } c' (\text{Suc last + x} - \text{last}))\)

by blast
have \(c\text{-undo-0}: c\text{-undo 0} = c_0\)
by (auto simp add: c-undo c-0)
from sequence-traces [OF trace-last trace-c'_first, simplified c-last, OF c'_first c-undo]
have trace-undo: trace c-undo 0 (last + k).

obtain u-ts u-shared u-m where
\(c\text{-undo-n}: \text{c-undo n} = (u-ts,u-m, u-shared)\)
by (cases c-undo n)
with last-bound c'_first c-last
have \(c'\text{-suc: } c' (\text{Suc n}) = (u-ts,u-m, u-shared)\)
apply (auto simp add: c-undo split: if-split-asm)
apply (subgoal-tac n=last)
apply auto
done

show ?thesis

proof (cases \(k < n - \text{last}\))
case True
with \(c'\text{-unsafe have unsafe: }\neg \text{safe-delayed (} c\text{-undo (last + k))}\)
by (auto simp add: c-undo c-last c'_first)
from have last + k \(\leq n\)
by auto
from safe-delayed-reach-inter.safe-config [OF this trace-undo, of last + k]
have safe-delayed (c-undo (last + k))

105
by (auto simp add: c-undo c-0)
with unsafe have False by simp
thus thesis ..

next

case False
with k-bound have k: k = n - last
by auto
have eq': Suc (last + (n - last)) = Suc n
  using last-bound
by simp
from c'-unch [rule-format, of k, simplified k eq', OF - c-suc c'-suc]
obtain u-ts-j: u-ts!j = ts!j and
  shared-unch: \( \forall a \in \mathcal{O}_t \). u-shared a = \mathcal{S}_t a \text{ and }
  shared-orig-unch: \( \forall a \in \mathcal{O}_t \). \mathcal{S} a = \mathcal{S}_t a \text{ and }
  mem-unch: \( \forall a \in \mathcal{O}_t \). u-m a = m_l a \text{ and }
  mem-unch-orig: \( \forall a \in \mathcal{O}_t \). m_a' = (m(a := f \theta_l)) a'
by auto

from c'-sim [rule-format, of k, simplified k eq', OF - c-suc c'-suc] i-bound neq-j-i
obtain u-ts-i: u-ts!i = ts!i and
  shared-sim: \( \forall a \notin \mathcal{O}_t \). u-shared a = \mathcal{S} a \text{ and }
  mem-sim: \( \forall a \notin \mathcal{O}_t \). u-m a = m a
by auto

from c'-leq [rule-format, of k] c'-suc c-suc
have leq-u-ts: length u-ts = length ts
  by (auto simp add: eq'k)

from j-bound leq-u-ts
have j-bound-u: j < length u-ts
  by simp
from i-bound leq-u-ts
have i-bound-u: i < length u-ts
  by simp
from k last-bound have l-k-eq: last + k = n
  by auto
from safe-delayed-reach-inter.safe-config [OF - trace-undo, simplified l-k-eq]
  k c-0 last-bound
have safe-delayed-c-undo': \( \forall x \leq n. \) safe-delayed (c-undo x)
  by (auto simp add: c-undo split: if-split-asm)
hence safe-delayed-c-undo: \( \forall x < n. \) safe-delayed (c-undo x)
  by auto
from trace-preserves-simple-ownership-distinct [OF - trace-undo,
  simplified l-k-eq c-undo-0, simplified, OF dist this, of n] dist c-undo-n
have dist-u-ts: simple-ownership-distinct u-ts
  by auto
then interpret dist-u-ts-inter: simple-ownership-distinct u-ts .

{ fix a

106
have \ u\text{-}shared\ a = \mathcal{S} a
proof (cases \ a \in \mathcal{O}_l)
case True with shared-unch
have \ u\text{-}shared\ a = \mathcal{S}_1 a
by auto
moreover
from True shared-orig-unch
have \mathcal{S} a = \mathcal{S}_1 a
by auto
ultimately show \ ?\text{thesis} by simp
next
case False
with shared-sim
show \ ?\text{thesis}
by auto
qed
}\ hence u\text{-}shared-eq: \ u\text{-}shared = \mathcal{S} by \ -(\text{rule ext, auto})

{ 
assume safe: map owned u\text{-}ts, map released u\text{-}ts, i \vdash (is, \emptyset, u\text{-}m, D, O, u\text{-}shared) \sqrt
then have False
proof cases
case Read
then show \ ?\text{thesis}
using\ ts-i ts-l-j race is j\text{-}bound i\text{-}bound u\text{-}ts-i u\text{-}ts-j leq-u\text{-}ts neq-j-i ts-j
by (auto simp add:eqs races-def split: if-split-asm)
next
case WriteNonVolatile
then show \ ?\text{thesis}
using\ ts-i ts-l-j race is j\text{-}bound i\text{-}bound u\text{-}ts-i u\text{-}ts-j leq-u\text{-}ts neq-j-i ts-j
by (auto simp add:eqs races-def split: if-split-asm)
next
case WriteVolatile
then show \ ?\text{thesis}
using\ ts-i ts-l-j race is j\text{-}bound i\text{-}bound u\text{-}ts-i u\text{-}ts-j leq-u\text{-}ts neq-j-i ts-j
apply (auto simp add:eqs races-def split: if-split-asm)
apply fastforce
done
next
case Fence
then show \ ?\text{thesis}
using\ ts-i ts-l-j race is j\text{-}bound i\text{-}bound u\text{-}ts-i u\text{-}ts-j leq-u\text{-}ts neq-j-i ts-j
by (auto simp add:eqs races-def split: if-split-asm)
next
case Ghost
then show \ ?\text{thesis}
using\ ts-i ts-l-j race is j\text{-}bound i\text{-}bound u\text{-}ts-i u\text{-}ts-j leq-u\text{-}ts neq-j-i ts-j
apply (auto simp add:eqs races-def split: if-split-asm)
apply fastforce
done

107
next
case (RMWReadOnly cond t a’ D f ret A L R W)
with ts-i is obtain
ins: ins = RMW a’ t (D, f) cond ret A L R W and
owned-or-shared: a’ ∈ O ∨ a’ ∈ dom u-shared and
cond: ¬ cond (θ(t ↦ u-m a’)) and
rels-race: ∀ j<length (map owned u-ts). i ≠ j → ((map released u-ts) ! j)
a’ ≠ Some False
by auto
from dist-u-ts-inter.simple-ownership-distinct [OF j-bound-u i-bound-u neq-j-i u-ts-j [simplified ts-i]]
have dist: O |_ O = {}
by auto
from owned-or-shared dist a-owned a-unshared shared-orig-unch
have a’-a: a’≠a
by (auto simp add: u-shared-eq domIff)
have u-m-eq: u-m a’ = m a’
proof (cases a’ ∈ O)
case True with mem-unch
have u-m a’ = m_i a’
by auto
moreover
from True mem-unch-orig a’-a
have m a’ = m_i a’
by auto
ultimately show ?thesis by simp
next
case False
with mem-sim
show ?thesis
by auto
qed
with ins cond rels-race show ?thesis
using ts-i ts-i-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j
by (auto simp add: eqs races-def u-shared-eq u-m-eq split: if-split-asm)
next
case (RMWWrite cond t a’ A L R D f ret W)
with ts-i is obtain
ins: ins = RMW a’ t (D, f) cond ret A L R W and
cond: cond (θ(t ↦ u-m a’)) and
a’: ∀ j<length (map owned u-ts). i ≠ j → a’ ≠ (map owned u-ts) ! j ∪ dom
((map released u-ts) ! j) and
safety:
A ⊆ dom u-shared ∪ O L ⊆ A R ⊆ O A ∩ R = {}
∀ j<length (map owned u-ts). i ≠ j → A ∩ ((map owned u-ts) ! j ∪ dom
((map released u-ts) ! j)) = {}
a’ ≠ read-only u-shared
by auto
from a’[rule-format, of j] j-bound-u u-ts-j ts-i-j neq-j-i
have \( a' \notin O' \)
by auto
from mem-sim [rule-format, OF this]

have \( u\text{-m-eq: } u\text{-m } a' = m a' \)
by auto

with ins cond safety \( a' \) show \(?\)thesis
using \( \text{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \)
apply (auto simp add:eqs races-def u-shared-eq u-m-eq split: if-split-asm)
apply fastforce
done
next
case Nil
then show \(?\)thesis
using \( \text{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \)
by (auto simp add:eqs races-def split: if-split-asm)
qed
}

hence \( \neg\) safe-delayed (u-ts, u-m, u-shared)
apply (clarsimp simp add: safe-delayed-def)
apply (rule-tac x=i in exI)
using u-ts-i ts-i i-bound-u
apply auto
done
moreover
from safe-delayed-c-undo' [rule-format, of n] c-undo-n
have safe-delayed (u-ts, u-m, u-shared)
by simp
ultimately have False
by simp
thus \(?\)thesis
by simp
qed
next
case WriteVolatile
with \( ts'\text{-j } ts' \text{-j-bound} \) have \( \mathcal{R}_j = \text{Map.empty} \)
by auto
with \( \mathcal{R}_j\text{-non-empty} \) have False by auto
thus \(?\)thesis ..
next
case Fence
with \( ts'\text{-j } ts' \text{-j-bound} \) have \( \mathcal{R}_j = \text{Map.empty} \)
by auto
with \( \mathcal{R}_j\text{-non-empty} \) have False by auto
thus \(?\)thesis ..
next
case RMWReadOnly
with \( ts'\text{-j } ts' \text{-j-bound} \) have \( \mathcal{R}_j = \text{Map.empty} \)
by auto
with \( \mathcal{R}_j\text{-non-empty} \) have False by auto

109
thus thesis ..

next
case RMWWrite
with ts\_j ts\_j ts\_j j-bound\_j have R\_j = Map.empty
by auto
with R\_j-non-empty have False by auto
thus thesis ..

next
case (Ghost A L R W)
then obtain
is\_j: is\_j = Ghost A L R W # is\_j\' and
\( \theta \_j \): \( \theta \_j \) and
sb\_j: sb\_j = sb\_j and
D\_j: D\_j = D\_j and
O\_j: O\_j = O\_j \cup A - R and
R\_j: R\_j = \text{augment-rels} \ (\text{dom } S\_j) \ R \ R\_j and
S\_j: S\_j = S\_j + W \ R \ R\_j L and
m\_j: m\_j = m\_j
by auto

note eqs\_j = \( \theta \_j \) sb\_j D\_j O\_j R\_j S\_j m\_j
from ts\_j ts\_j ts\_j j-bound\_j eqs\_j
obtain eqs: p\_i = p\_j is\_j = is\_j \( \theta \_i = \theta \_j \)
\( D\_i = D\_j \) O\_i \cup A - R = R = O\_j
\( \text{augment-rels} \ (\text{dom } S\_i) \ R \ R\_i = R\_j \)
by auto

from safe\_i [simplified is\_i]
obtain
A-shared-owned: A \subseteq \text{dom } S\_i \cup O\_i and L-A: L \subseteq A and R-owns: R \subseteq O\_i and
A-R: A \cap R = \{\} and
\( \forall j' < \text{length} \ (\text{map owned } ts\_i \text{). } j \neq j' \longrightarrow A \cap (\text{map owned } ts\_i|j' \cup \text{dom } (\text{map released } ts\_i)\_j') = \{\} \)
by cases auto

from A-shared-owned L-A R-owns A-R
have shared-eq: \( \forall a. \ a \notin O\_i \longrightarrow a \notin O\_i' \longrightarrow S\_i \ a = (S\_i + W \ R \ R\_i L) \ a \)
by (auto simp add: restrict-shared-def augment-shared-def O\_i' split: option.splits)

from undo-local-steps [where c=c, OF trace-rest c-last' idle-rest' safe-delayed-last, simplified ts\_i', simplified]
OF j-bound\_i ts\_i-j [simplified], simplified m\_i' S\_i', simplified,
OF shared-eq dist-last dist-last' [simplified ts\_i', simplified] safe-delayed-upto-last

obtain c' k where
k-bound: k \leq n - last and
trace-c': trace c' (Suc last) k and
c'-first: c' (Suc last) = (ts\_i, m\_i, S\_i) and
c'-leq: (\( \forall x \leq k. \ \text{length} \ \text{fst} \ (c' (Suc (last + x)))) = \text{length} \ \text{fst} \ (c (Suc (last + x))) \))
and
c'-safe: \((\forall x < k. \text{ safe-delayed } (c' (\text{Suc} \ (\text{last} + x))))\) and \\
c'-unsafe: \((k < n - \text{last} \implies \neg \text{safe-delayed } (c' (\text{Suc} \ (\text{last} + k))))\) and \\
c'-unch: \\
\((\forall x \leq k. \forall ts_x S_x m_x. \c (\text{Suc} \ (\text{last} + x)) = (ts_x, m_x, S_x) \implies \\
(\forall ts_x', S_x' m_x'. \c' (\text{Suc} \ (\text{last} + x)) = (ts_x', m_x', S_x') \implies \\
ts_x'! j = ts_l! j \land \\
(\forall a \in \mathcal{O}_l, S_x' a = S_l a) \land \\
(\forall a \in \mathcal{O}_l, S_x a = (S_l \oplus_{\mathbb{W}} R \oplus_{\mathbb{A}} L) a) \land \\
(\forall a \in \mathcal{O}_l, m_x' a = m_l a) \land (\forall a' \in \mathcal{O}_l, m_x a' = (m_l) a')))\) and \\
c'-sim: \\
\((\forall x \leq k. \forall ts_x S_x m_x. \c (\text{Suc} \ (\text{last} + x)) = (ts_x, m_x, S_x) \implies \\
(\forall ts_x', S_x' m_x'. \c' (\text{Suc} \ (\text{last} + x)) = (ts_x', m_x', S_x') \implies \\
(\forall j \in \text{length } ts_x, j \neq j \implies ts_x'! j = ts_l! j) \land \\
(\forall a \in \mathcal{O}_l, a \notin \mathcal{O}_l \implies S_x' a = S_x a) \land \\
(\forall a \in \mathcal{O}_l, m_x' a = m_x a))\))

by (clarsimp )

obtain c-undo where c-undo: \(c\text{-undo} = (\lambda x. \text{if } x \leq \text{last} \text{ then } c x \text{ else } c' (\text{Suc } (\text{last} + x) \text{ - last} ))\)

by blast

have c-undo-0: c-undo 0 = c_0

by (auto simp add: c-undo c-0)

from sequence-traces [OF trace-last trace-c', simplified c-last, OF c'-first c-undo]

have trace-undo: trace c-undo 0 (last + k) .

obtain u-ts u-shared u-m where \\
c-undo-n: c-undo n = (u-ts,u-m, u-shared)

by (cases c-undo n)

with last-bound c'-first c-last

have c'-suc: \(c' (\text{Suc} n) = (u-ts,u-m, u-shared)\)

apply (auto simp add: c-undo split: if-split-asm)

apply (subgoal-tac n=last)

apply auto

done

show \?thesis

proof (cases k < n - last)

case True

with c'-unsafe have unsafe: \(\neg \text{safe-delayed } (c\text{-undo } (\text{last} + k))\)

by (auto simp add: c-undo c-last c'-first)

from True have last + k \leq n

by auto

from safe-delayed-reach-inter.safe-config [OF this trace-undo, of last + k]

have safe-delayed (c-undo (last + k))

by (auto simp add: c-undo c-0)

with unsafe have False by simp
thus \( ? \)\thesis ..

next

\begin{itemize}
  \item \textbf{case} False
  \item \textbf{with} \( k \)-bound \textbf{have} \( k: k = n \) \(-\) last
    \item \textbf{by} auto
  \item \textbf{have} \( eq': \text{Suc} \ (last + (n - last)) = \text{Suc} \ n \)
    \item \textbf{using} last-bound
    \item \textbf{by} simp
  \item \textbf{from} \( c'\)-unch \ [\text{rule-format, of} \ k, \ simplified \ k \ eq', \ \text{OF} - \ c\text{-suc} \ c'\text{-suc}] \)
    \textbf{obtain} \( u\text{-ts}-j: u\text{-ts}!j = ts!j \) \textbf{and}
      \item \textbf{shared-unch:} \( \forall a \in O_1, \ u\text{-shared} \ a = S_l \ a \) \textbf{and}
      \item \textbf{shared-orig-unch:} \( \forall a \in O_1, \ S \ a = (S_l \ominus W \ominus R \ominus A \ominus L) \ a \) \textbf{and}
      \item \textbf{mem-unch:} \( \forall a \in O_1, \ u\text{-m} \ a = m_l \ a \) \textbf{and}
      \item \textbf{mem-unch-orig:} \( \forall a' \in O_1, \ m \ a' = m_l \ a' \)
    \item \textbf{by} auto
  \item \textbf{from} \( c'\)-sim \ [\text{rule-format, of} \ k, \ simplified \ k \ eq', \ \text{OF} - \ c\text{-suc} \ c'\text{-suc}] \) \( i\)-bound \( neq\)-j-i
    \textbf{obtain} \( u\text{-ts}-i: u\text{-ts}!i = ts!i \) \textbf{and}
      \item \textbf{shared-sim:} \( \forall a. \ a \neq O_1 \rightarrow a \neq O_1' \rightarrow u\text{-shared} \ a = S \ a \) \textbf{and}
      \item \textbf{mem-sim:} \( \forall a. \ a \neq O_1 \rightarrow u\text{-m} \ a = m \ a \)
    \item \textbf{by} auto
  \item \textbf{from} \( c'\)-leq \ [\text{rule-format, of} \ k] \ c'\text{-suc} \ c\text{-suc} \)
    \textbf{have} \( \text{leq-u-ts}: \text{length} \ u\text{-ts} = \text{length} \ ts \)
    \item \textbf{by} (auto simp add: eq' k)
  \item \textbf{from} \( j\)-bound \( \text{leq-u-ts} \)
    \textbf{have} \( j\)-bound-u: \( j < \text{length} \ u\text{-ts} \)
    \item \textbf{by} simp
  \item \textbf{from} \( i\)-bound \( \text{leq-u-ts} \)
    \textbf{have} \( i\)-bound-u: \( i < \text{length} \ u\text{-ts} \)
    \item \textbf{by} simp
  \item \textbf{from} \( k \) \( \text{last-bound} \)
    \textbf{have} \( l\text{-k-eq}: \text{last} + k = n \)
    \item \textbf{by} auto
  \item \textbf{from} \( \text{safe-delayed-reach-inter.safe-config} \ [\text{OF} - \text{trace-undo}, \ simplified \ l\text{-k-eq}] \)
    \( k \ c\text{-0 last-bound} \)
    \textbf{have} \( \text{safe-delayed-c-undo}': \forall x \leq n. \ \text{safe-delayed} \ (c\text{-undo} \ x) \)
      \item \textbf{by} (auto simp add: c-undo split: if-split-asm)
    \item hence \( \text{safe-delayed-c-undo}: \forall x < n. \ \text{safe-delayed} \ (c\text{-undo} \ x) \)
      \item \textbf{by} auto
  \item \textbf{from} \( \text{trace-preserves-simple-ownership-distinct} \ [\text{OF} - \text{trace-undo}, \ simplified \ l\text{-k-eq} \ c\text{-undo}-0, \ simplified, \ \text{OF} \ \text{dist this, of} \ n] \ \text{dist} \ c\text{-undo-n} \)
    \textbf{have} \( \text{dist-u-ts: simple-ownership-distinct} \ u\text{-ts} \)
      \item \textbf{by} auto
  \item \textbf{then interpret} \( \text{dist-u-ts-inter: simple-ownership-distinct} \ u\text{-ts} \)
    \item \{
      \item \textbf{fix} \( a \)
      \item \textbf{have} \( u\text{-m} \ a = m \ a \)
      \item \textbf{proof} (cases \( a \in O_1 \))
        \item \textbf{case} True \textbf{ with} \text{mem-unch}
have u-m a = m_{i} a
    by auto
moreover
from True mem-unch-orig
have m a = m_{i} a
    by auto
ultimately show ?thesis by simp
next
case False
with mem-sim
show ?thesis
    by auto
qed
}
hence u-m-eq: u-m = m by (rule ext, auto)
{
assume safe: map owned u-ts,map released u-ts,i LATIN=(is,\emptyset,u-m,D,O,u-shared)√
then have False
proof cases
  case (Read a volatile t)
  with ts-i is obtain
    ins: ins = Read volatile a t and
    access-cond: a \in O \lor a \in read-only u-shared \lor volatile \land a \in dom u-shared
and
    rels-cond: \forall j < length u-ts. i \neq j \longrightarrow ((map released u-ts) ! j) a \neq Some
False and
rels-non-volatile-cond: \neg volatile \longrightarrow (\forall j < length u-ts. i \neq j \longrightarrow a \not\in dom ((map released u-ts) ! j)) and
    clean: volatile \longrightarrow \neg D
    by auto

from race ts-j
have rc: augment-rels (dom S_{i}) R R_{i} a = Some False \lor
    (\neg volatile \land a \in dom (augment-rels (dom S_{i}) R R_{i}))
    by (auto simp add: races-def ins eqs)
from rels-cond [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]]
  u-ts-j ts_{i-j} j-bound-u
have R_{i-a}: R_{i} a \neq Some False
    by auto
  from dist-u-ts-inter.simple-ownership-distinct [OF j-bound-u i-bound-u neq-j-i u-ts-j [simplified ts_{i-j}]
    u-ts-i [simplified ts_{i}]]
have dist: O_{i} \cap O = \{\}
    by auto

show ?thesis
proof (cases volatile)
case True
  note volatile=this
show ?thesis
proof (cases a \in R)
case False
  with rc \( R \)-a show False
  by (auto simp add: augment-rels-def volatile)
next
  case True
  with R-owns
  have a-owns\(_l\): a \( \in \) \( O \)_\( l \)
  by auto
from shared-unch [rule-format, OF a-owns\(_l\)]
have u-shared-eq: u-shared a = \( S \)_\( l \) a
  by auto
from a-owns\(_l\) dist have a \( \notin \) \( O \)
  by auto
moreover
{ assume a \( \in \) read-only u-shared
  with u-shared-eq have \( S \)_\( l \) a = Some False
  by (auto simp add: read-only-def)
  with rc True \( R \)-a have False
  by (auto simp add: augment-rels-def split: option.splits simp add: domIff volatile)
} moreover
{ assume a \( \in \) dom u-shared
  with u-shared-eq rc True \( R \)-a have False
  by (auto simp add: augment-rels-def split: option.splits simp add: domIff volatile)
} ultimately show False
using access-cond
by auto
qed
next
  case False
  note non-volatile = this
  from rels-non-volatile-cond [rule-format, OF False j-bound-u neq-j-i [symmetric]] u-ts-j ts\(_{i-j}\) j-bound-u
  have \( R \)_\( t \)-a: \( R \)_\( l \) a = None
  by (auto simp add: domIff)
show ?thesis
proof (cases a \( \in \) R)
  case False
  with rc \( R \)_\( t \)-a show False
  by (auto simp add: augment-rels-def non-volatile domIff)
next
  case True
  with R-owns
  have a-owns\(_l\): a \( \in \) \( O \)_\( l \)
  by auto
from shared-unch [rule-format, OF a-owns l]
have u-shared-eq: u-shared a = $S_l$ a
  by auto
from a-owns l dist have a-unowned: a $\notin$ $O$
  by auto
moreover
from ro-last-last interpret
read-only-unowned $S_l$ ts_i.
  from read-only-unowned [OF j-bound l ts_i-j] a-owns l have a-unsh: a $\notin$
read-only $S_l$ by auto
  
  assume a $\in$ read-only u-shared
  with u-shared-eq have sh: $S_l$ a = Some False
  by (auto simp add: read-only-def)

  with rc True /$R_l$-a access-cond u-shared-eq a-unowned sh a-owns l a-unsh
have False
  by (auto simp add: augment-rels-def split: option.splits simp add: domIff non-volatile read-only-def)
}

moreover
{
  assume a $\in$ dom u-shared
  with u-shared-eq rc True /$R_l$-a a-owns l a-unsh access-cond dist have
False
  by (auto simp add: augment-rels-def split: option.splits simp add: domIff non-volatile read-only-def)
}

ultimately show False
using access-cond
by (auto)
qed
qed
next
case (WriteNonVolatile a D f A' L' R' W')
with ts-i is obtain
ins: ins = Write False a (D, f) A’ L’ R’ W’
  and a-owned: a $\in$ $O$
  and a-unshared: a $\notin$ dom u-shared
  and a-unreleased: $\forall j < \text{length } u-ts. i \neq j \rightarrow a \notin \text{dom } (\text{map released } u-ts)! j$
  by auto
from dist-u-ts-inter.simple-ownership-distinct [OF j-bound-u i-bound-u neq-j-i u-ts-j [simplified ts_i-j]]
  u-ts-i [simplified ts-i]]
have dist: $O_l \cap O = \{\}$
  by auto
from race ts-j
have rc: a $\in$ dom (augment-rels (dom $S_l$) R $R_l$)
  by (auto simp add: races-def ins eqs)
from a-unreleased [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]]
u-ts-j ts_i-j j-bound-u
have $R_l$-a: $a \notin \text{dom } R_l$
by auto
show False

proof (cases $a \in R$)
case False
  with $\text{rc } R_l$-a show False
    by (auto simp add: augment-rels-def domIff)
next
case True
  with $R$-owns
  have a-owns: $a \in O_l$
    by auto
  with a-owned dist show False
    by auto
qed

next
case (WriteVolatile $a$ $A'$ $L'$ $R'$ $D$ $f$ $W'$)
  with ts-i is obtain
    ins: ins = Write True $a$ ($D$, $f$) $A'$ $L'$ $R'$ $W'$
    and
    a-un-owned-released: $\forall j < \text{length u-ts}. \ i \neq j \rightarrow \ a \notin ((\text{map owned u-ts}) \ j)$
    and $A'$-owns-shared: $A' \subseteq \text{dom u-shared} \cup O$
    and $L'$-$A'$: $L' \subseteq A'$
    and $R'$-owned: $R' \subseteq O$
    and $A'$-$R'$: $A' \cap R' = \{}$
    and
    acq-ok: $\forall j < \text{length u-ts}. \ i \neq j \rightarrow A' \cap ((\text{map owned u-ts}) \ j \cup \text{dom ((map released u-ts))}) = \{\}$
    and
writeable: $a \notin \text{read-only u-shared}$
by auto
  from a-un-owned-released [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]] u-ts-j ts-i-j j-bound-u
  obtain $O_l$-a: $a \notin O_l$ and $R_l$-a: $a \notin \text{dom } (R_l)$
  by auto
  from acq-ok [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]] u-ts-j ts-i-j j-bound-u
  obtain $O_l$-$A'$: $A' \cap O_l = \{}$ and $R_l$-$A'$: $A' \cap \text{dom } (R_l) = \{}$
  by auto
  {
    assume rc: $a \in \text{dom (augment-rels (dom } S_i) \ R_i)$
    have False
    proof (cases $a \in R$)
      case False
        with $\text{rc } R_l$-a show False
          by (auto simp add: augment-rels-def domIff)
      next
      case True
        with $R$-owns
        have a-owns: $a \in O_l$
          by auto
  }
with $O_\Gamma$-a show False
  by auto
qed
}

moreover
{
  assume rc: $A' \cap \text{dom (augment-rels (dom S) R R)} \neq \{\}
  then obtain a' where a'-A': a' \in A' and a'-aug: a' \in \text{dom (augment-rels (dom S) R R)}$
  by auto
  have False
proof (cases a' \in R)
  case False
  with a'-aug a'-A' $\mathcal{R}_\Gamma$-A' show False
  by (auto simp add: augment-rels-def domIff)
next
  case True
  with R-owns have a'-owns: a' \in $O_\Gamma$
  by auto
  with $O_\Gamma$-A' a'-A' show False
  by auto
qed
}

ultimately show False
using race ts-j
  by (auto simp add: races-def ins eqs)

next

  case Fence
  then show ?thesis
using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j
  by (auto simp add: eqs races-def split: if-split-asm)

next

  case (Ghost A'L'R'W')
  with ts-i is obtain
    ins: ins = Ghost A'L'R'W' and
    A'-owns-shared: A' \subseteq \text{dom u-shared } \cup O and
    L'-A': L' \subseteq A' and
    R'-owned: R' \subseteq O and
    A'R': A' \cap R' = \{\} and
    acq-ok: \forall j<\text{length u-ts}. i \neq j \rightarrow A' \cap ((\text{map owned u-ts}) ! j \cup \text{dom ((map released u-ts)} ! j)) = \{\}
  by auto
  from acq-ok [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]] u-ts-j
  ts-j j-bound-u
  obtain $O_\Gamma$-A': $A' \cap O_\Gamma = \{\} and $\mathcal{R}_\Gamma$-A': $A' \cap \text{dom (R)} = \{\}
  by auto

from race ts-j
obtain a' where a'-A': a' \in A' and
  a'-aug: a' \in \text{dom (augment-rels (dom S) R R)}

117
by (auto simp add: races-def ins eqs)
show False
proof (cases a' ∈ R)
case False
with a'-aug a'-A' R-A' show False
  by (auto simp add: augment-rels-def domIff)
next
case True
with R-owns have a'-owns$: a' ∈ O₁
  by auto
with O₁-A' a'-A' show False
  by auto
qed
next
case (RMWReadOnly cond t a D f ret A' L' R' W')
with ts-i is obtain
  ins: ins = RMW a t (D, f) cond ret A' L' R' W' and
  owned-or-shared: a ∈ O ∨ a ∈ dom u-shared and
  cond: ¬ cond (θ(t ↦ → u-m a)) and
  rels-race: ∀ j < length (map owned u-ts). i ≠ j ⟹ ((map released u-ts) ! j)
  a ≠ Some False
  by auto
  from dist-u-ts-inter.simple-ownership-distinct [OF j-bound-u i-bound-u
neq-j-i u-ts-j] [simplified ts-i]
    u-ts-i [simplified ts-i]]
  have dist: O₁ ∩ O = {}
    by auto
  from race ts-j cond
  have rc: augment-rels (dom S_l) R R_l a = Some False
    by (auto simp add: races-def ins eqs u-m-eq)
  from rels-race [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]]
    u-ts-j ts-i j-bound-u
  have R_l-a: R_l a ≠ Some False
    by auto
show ?thesis
proof (cases a ∈ R)
case False
with rc R_l-a show False
  by (auto simp add: augment-rels-def)
next
case True
with R-owns
  have a-owns$: a ∈ O₁
    by auto
  from shared-unch [rule-format, OF a-owns]$ have u-shared-eq: u-shared a = S_l a
    by auto
  from a-owns$ dist have a ∉ O

118
by auto
with \texttt{u-shared-eq rc True }\mathcal{R}_l\text{-a owned-or-shared show False }
by (auto simp add: augment-rels-def split: option.splits simp add: domIff) qed

next
case (RMWWrite cond t a A' L' R' D f ret W')
with ts-i is obtain
ins: ins = RMW a t (D, f) cond ret A' L' R' W' and
cond: cond (\(\theta(t \mapsto u-m \text{ a})\)) and
a-un-owned-released: \(\forall j < \text{length (map owned u-ts)}\). i \neq j \rightarrow a \notin (\text{map owned u-ts})!j\) and
A'-owns-shared:A' \subseteq \text{dom u-shared} \cup O and
L'-A': L' \subseteq A' and
R'-owned: R' \subseteq O and
A'-R': A' \cap R' = {} and
acq-ok: \(\forall j < \text{length (map owned u-ts)}\). i \neq j \rightarrow A' \cap ((\text{map owned u-ts})!j \cup \text{dom ((map released u-ts)}!j) = {} and
writeable: a \notin \text{read-only u-shared}
by auto

from a-un-owned-released [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]] u-ts-j ts_i-j j-bound-u
obtain O_i-a: a \notin O_i and \mathcal{R}_l\text{-a: a }\notin\text{ dom }\mathcal{R}_i
by auto
from acq-ok [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]] u-ts-j ts_i-j j-bound-u
obtain O_i-A': A' \cap O_i = {} and \mathcal{R}_l\text{-A': A' \cap dom }\mathcal{R}_i = {}
by auto
{
assume rc: a \in \text{dom (augment-rels (dom }\mathcal{S}_{l_1} ) R \mathcal{R}_l) 

have False

proof (cases a \in R)

  case False

    with rc \mathcal{R}_l\text{-a show False }

    by (auto simp add: augment-rels-def domIff)

next
  case True

  with R-owns

  have a-owns_i: a \in O_i

  by auto

  with O_i-a show False

  by auto

qed

}

moreover
{
assume rc: A' \cap \text{dom (augment-rels (dom }\mathcal{S}_{l_1} ) R \mathcal{R}_l) \neq \{\}

then obtain a' where a'-A': a' \in A' and a'-aug: a' \in \text{dom (augment-rels (dom }\mathcal{S}_{l_1} ) R \mathcal{R}_l)
by auto
have False
proof (cases a' ∈ R)
  case False
  with a'-aug a'-A' R-A' show False
    by (auto simp add: augment-rels-def domIff)
next
  case True
  with R-owns have a'-owns_l: a' ∈ O_l
    by auto
  with O_l-A' a'-A' show False
    by auto
qed
}
ultimately show False
using race ts-j cond
  by (auto simp add: races-def ins eqs u-m-eq)
next
next
  case Nil
  then show ?thesis
    using ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j
    by (auto simp add: eqs races-def split: if-split-asm)
qed
hence ¬ safe-delayed (u-ts, u-m, u-shared)
  apply (clarsimp simp add: safe-delayed-def)
  apply (rule-tac x=i in exI)
  using u-ts-i ts-i i-bound-u
  apply auto
  done
moreover
from safe-delayed-c-undo' [rule-format, of n] c-undo-n
have safe-delayed (u-ts, u-m, u-shared)
  by simp
ultimately have False
  by simp
  thus ?thesis
  by simp
qed
qed
next
  case (StoreBuffer - p is ∅ sb D O R sb' O' R')
hence False
  by (auto simp add: empty-storebuffer-step-def)
  thus ?thesis ..
qed
qed
}
ultimately show ?thesis
using last-action-of-thread [where i=j, OF trace]
by blast
qed
qed

datatype 'p memref =
  Write\sb bool addr sop val acq lcl rel wrt
| Read\sb bool addr tmp val
| Prog\sb 'p 'p instrs
| Ghost\sb acq lcl rel wrt

type-synonym 'p store-buffer = 'p memref list

inductive flush-step:: memory × 'p store-buffer × owns × rels × shared ⇒ memory × 'p store-buffer × owns × rels × shared ⇒ bool
(- ⇒ [60,60] 100)

where
Write\sb: [O' = (if volatile then O ∪ A − R else O);
S' = (if volatile then S ⊕ W R ⊖ A L else S);
R'=(if volatile then Map.empty else R)]
⇒

(m, Write\sb volatile a sop v A L R W# rs,O,R,S) →\(f\) (m(a := v), rs,O,R',S')

| Read\sb: (m, Read\sb volatile a t v#rs,O,R,S) →\(f\) (m, rs,O,R, S)
| Prog\sb: (m, Prog\sb p p' is#rs,O,R, S) →\(f\) (m, rs,O,R, S)
| Ghost: (m, Ghost\sb A L R W# rs,O,R,S) →\(f\) (m, rs,O ∪ A − R, augment-rels (dom S) R R, S ⊕ W R ⊖ A L )

abbreviation flush-steps::memory × 'p store-buffer × owns × rels × shared ⇒ memory × 'p store-buffer × owns × rels × shared⇒ bool
(- ⇒ [-60,60] 100)

where
flush-steps == flush-step^**

term x →\(f^*\) Y

lemmas flush-step-induct =
  flush-step.induct [split-format (complete),
  consumes 1, case-names Write\sb Read\sb Prog\sb Ghost]

inductive store-buffer-step:: memory × 'p store-buffer × 'owns × 'rels × 'shared ⇒ memory × 'p memref list × 'owns × 'rels × 'shared ⇒ bool
(- ⇒w [-60,60] 100)

where
SBWrite\sb:
  (m, Write\sb volatile a sop v A L R W# rs,O,R,S) →w (m(a := v), rs,O,R,S)

abbreviation store-buffer-steps::memory × 'p store-buffer × 'owns × 'rels × 'shared ⇒ memory × 'p store-buffer × 'owns × 'rels × 'shared⇒ bool
(- →w^* [-60,60] 100)

where
store-buffer-steps == store-buffer-step^**
\textbf{term} \( x \rightarrow w^* \ Y \)

\textbf{fun} buffered-val :: \( 'p \text{ memref list} \Rightarrow \text{addr} \Rightarrow \text{val option} \)
\textbf{where}
  \begin{align*}
  \text{buffered-val} \ [] \ a &= \text{None} \\
  \text{buffered-val} \ (r \# \ rs) \ a' &= \\
  \text{(case } r \text{ of} \\
  \quad \text{Write}_{sb} \ \text{volatile} \ a \ v \ - \ - \ - &\Rightarrow \text{(case } \text{buffered-val} \ \text{rs} \ a' \text{ of} \\
  \quad \quad \text{None } \Rightarrow \text{(if } a' = \text{a then Some } v \text{ else None)} \\
  \quad \quad \text{Some } v' \Rightarrow \text{Some } v') \\
  \quad \text{- } &\Rightarrow \text{buffered-val} \ \text{rs} \ a')
  \end{align*}

\textbf{definition} address-of :: \( 'p \text{ memref } \Rightarrow \text{addr set} \)
\textbf{where}
\begin{align*}
\text{address-of} \ r &= \text{(case } r \text{ of} \\
  \quad \text{Write}_{sb} \ \text{volatile} \ a \ v \ - \ - \ - &\Rightarrow \{\text{a}\} | \text{Read}_{sb} \ \text{volatile} \ a \ t \ v \Rightarrow \{\text{a}\} | \\
  \quad \text{- } &\Rightarrow \{\}
  \end{align*}

\textbf{lemma} address-of-simps \[\text{simp}\]:
\begin{align*}
\text{address-of} \ (\text{Write}_{sb} \ \text{volatile} \ a \ \text{sop} \ v \ A \ L \ R \ W) &= \{\text{a}\} \\
\text{address-of} \ (\text{Read}_{sb} \ \text{volatile} \ a \ t \ v) &= \{\text{a}\} \\
\text{address-of} \ (\text{Prog}_{sb} \ p \ p' \ \text{is}) &= \{\} \\
\text{address-of} \ (\text{Ghost}_{sb} \ A \ L \ R \ W) &= \{\} \\
\text{by} \ (\text{auto simp add: address-of-def})
\end{align*}

\textbf{definition} is-volatile :: \( 'p \text{ memref } \Rightarrow \text{bool} \)
\textbf{where}
\begin{align*}
\text{is-volatile} \ r &= \text{(case } r \text{ of} \\
  \quad \text{Write}_{sb} \ \text{volatile} \ a \ v \ - \ - \ - &\Rightarrow \text{volatile} | \text{Read}_{sb} \ \text{volatile} \ a \ t \ v \Rightarrow \text{volatile} \\
  \quad \text{- } &\Rightarrow \text{False})
  \end{align*}

\textbf{lemma} is-volatile-simps \[\text{simp}\]:
\begin{align*}
\text{is-volatile} \ (\text{Write}_{sb} \ \text{volatile} \ a \ \text{sop} \ v \ A \ L \ R \ W) &= \text{volatile} \\
\text{is-volatile} \ (\text{Read}_{sb} \ \text{volatile} \ a \ t \ v) &= \text{volatile} \\
\text{is-volatile} \ (\text{Prog}_{sb} \ p \ p' \ \text{is}) &= \text{False} \\
\text{is-volatile} \ (\text{Ghost}_{sb} \ A \ L \ R \ W) &= \text{False} \\
\text{by} \ (\text{auto simp add: is-volatile-def})
\end{align*}

\textbf{definition} is-Write\(_{sb}\) :: \( 'p \text{ memref } \Rightarrow \text{bool} \)
\textbf{where}
\begin{align*}
\text{is-Write}\(_{sb}\) \ r &= \text{(case } r \text{ of} \\
  \quad \text{Write}_{sb} \ \text{volatile} \ a \ v \ - \ - \ - &\Rightarrow \text{True} | \text{- } &\Rightarrow \text{False})
  \end{align*}

\textbf{definition} is-Read\(_{sb}\) :: \( 'p \text{ memref } \Rightarrow \text{bool} \)
\textbf{where}
\begin{align*}
\text{is-Read}\(_{sb}\) \ r &= \text{(case } r \text{ of} \\
  \quad \text{Read}_{sb} \ \text{volatile} \ a \ t \ v &\Rightarrow \text{True} | \text{- } &\Rightarrow \text{False})
  \end{align*}

\textbf{definition} is-Prog\(_{sb}\) :: \( 'p \text{ memref } \Rightarrow \text{bool} \)
\textbf{where}
\begin{align*}
\text{is-Prog}\(_{sb}\) \ r &= \text{(case } r \text{ of} \\
  \quad \text{Prog}_{sb} \ - \ - \ - &\Rightarrow \text{True} | \text{- } &\Rightarrow \text{False})
  \end{align*}
**definition** is-Ghost\(sb:: (^p \text{memref} \Rightarrow \text{bool})

**where**

\(\text{is-Ghost}\(sb\ r = \text{(case } r \text{ of } \text{Ghost}\(sb\ - - - - \Rightarrow \text{True} \mid - \Rightarrow \text{False})\)

**lemma** is-Write\(sb\)-simps [simp]:

\(\text{is-Write}\(sb\ \text{(Write}\(sb\ \text{volatile } a \text{ sop } v \text{ A L R W}) = \text{True} \text{)}

\(\text{is-Write}\(sb\ \text{(Read}\(sb\ \text{volatile } a \text{ t } v) = \text{False} \text{)}

\(\text{is-Write}\(sb\ \text{(Prog}\(sb\ p p’ is) = \text{False} \text{)}

\(\text{is-Write}\(sb\ \text{(Ghost}\(sb\ A L R W) = \text{False} \text{)}

\text{by (auto simp add: is-Write}\(sb\-\text{def})\)

**lemma** is-Read\(sb\)-simps [simp]:

\(\text{is-Read}\(sb\ \text{(Read}\(sb\ \text{volatile } a \text{ t } v) = \text{True} \text{)}

\(\text{is-Read}\(sb\ \text{(Write}\(sb\ \text{volatile } a \text{ sop } v \text{ A L R W}) = \text{False} \text{)}

\(\text{is-Read}\(sb\ \text{(Prog}\(sb\ p p’ is) = \text{False} \text{)}

\(\text{is-Read}\(sb\ \text{(Ghost}\(sb\ A L R W) = \text{False} \text{)}

\text{by (auto simp add: is-Read}\(sb\-\text{def})\)

**lemma** is-Prog\(sb\)-simps [simp]:

\(\text{is-Prog}\(sb\ \text{(Read}\(sb\ \text{volatile } a \text{ t } v) = \text{False} \text{)}

\(\text{is-Prog}\(sb\ \text{(Write}\(sb\ \text{volatile } a \text{ sop } v \text{ A L R W}) = \text{False} \text{)}

\(\text{is-Prog}\(sb\ \text{(Prog}\(sb\ p p’ is) = \text{True} \text{)}

\(\text{is-Prog}\(sb\ \text{(Ghost}\(sb\ A L R W) = \text{False} \text{)}

\text{by (auto simp add: is-Prog}\(sb\-\text{def})\)

**lemma** is-Ghost\(sb\)-simps [simp]:

\(\text{is-Ghost}\(sb\ \text{(Read}\(sb\ \text{volatile } a \text{ t } v) = \text{False} \text{)}

\(\text{is-Ghost}\(sb\ \text{(Write}\(sb\ \text{volatile } a \text{ sop } v \text{ A L R W}) = \text{False} \text{)}

\(\text{is-Ghost}\(sb\ \text{(Prog}\(sb\ p p’ is) = \text{False} \text{)}

\(\text{is-Ghost}\(sb\ \text{(Ghost}\(sb\ A L R W) = \text{True} \text{)}

\text{by (auto simp add: is-Ghost}\(sb\-\text{def})\)

**definition** is-volatile-Write\(sb:: (^p \text{memref} \Rightarrow \text{bool})

**where**

\(\text{is-volatile-Write}\(sb\ r = \text{(case } r \text{ of } \text{Write}\(sb\ \text{volatile } a - v - - - - \Rightarrow \text{volatile} \mid - \Rightarrow \text{False})\)

**lemma** is-volatile-Write\(sb\)-simps [simp]:

\(\text{is-volatile-Write}\(sb\ \text{(Write}\(sb\ \text{volatile } a \text{ sop } v \text{ A L R W}) = \text{volatile} \text{)}

\(\text{is-volatile-Write}\(sb\ \text{(Read}\(sb\ \text{volatile } a \text{ t } v) = \text{False} \text{)}

\(\text{is-volatile-Write}\(sb\ \text{(Prog}\(sb\ p p’ is) = \text{False} \text{)}

\(\text{is-volatile-Write}\(sb\ \text{(Ghost}\(sb\ A L R W) = \text{False} \text{)}

\text{by (auto simp add: is-volatile-Write}\(sb\-\text{def})\)

**lemma** is-volatile-Write\(sb\)-address-of [simp]: is-volatile-Write\(sb\ x = \Rightarrow \text{address-of } x \neq \{\}

\text{by (cases } x \text{) auto}\)

**definition** is-volatile-Read\(sb:: (^p \text{memref} \Rightarrow \text{bool})

**where**

\(\text{is-volatile-Read}\(sb\ r = \text{(case } r \text{ of } \text{Read}\(sb\ \text{volatile } a \text{ t } v \Rightarrow \text{volatile} \mid - \Rightarrow \text{False})\)

123
lemma is-volatile-Read\sb-simps [simp]:
is-volatile-Read\sb (Read\sb volatile a t v) = volatile
is-volatile-Read\sb (Write\sb volatile a sop v A L R W) = False
is-volatile-Read\sb (Prog\sb p p' is) = False
is-volatile-Read\sb (Ghost\sb A L R W) = False
by (auto simp add: is-volatile-Read\sb-def)

definition is-non-volatile-Write\sb:: 'p memref ⇒ bool
where
is-non-volatile-Write\sb r = (case r of Write\sb volatile a - v - - - - ⇒ ¬ volatile | - ⇒ False)

lemma is-non-volatile-Write\sb-simps [simp]:
is-non-volatile-Write\sb (Write\sb volatile a sop v A L R W) = (¬ volatile)
is-non-volatile-Write\sb (Read\sb volatile a t v) = False
is-non-volatile-Write\sb (Prog\sb p p' is) = False
is-non-volatile-Write\sb (Ghost\sb A L R W) = False
by (auto simp add: is-non-volatile-Write\sb-def)

definition is-non-volatile-Read\sb:: 'p memref ⇒ bool
where
is-non-volatile-Read\sb r = (case r of Read\sb volatile a t v ⇒ ¬ volatile | - ⇒ False)

lemma is-non-volatile-Read\sb-simps [simp]:
is-non-volatile-Read\sb (Read\sb volatile a t v) = (¬ volatile)
is-non-volatile-Read\sb (Write\sb volatile a sop v A L R W) = False
is-non-volatile-Read\sb (Prog\sb p p' is) = False
is-non-volatile-Read\sb (Ghost\sb A L R W) = False
by (auto simp add: is-non-volatile-Read\sb-def)

lemma is-volatile-split: is-volatile r =
(is-volatile-Read\sb r ∨ is-volatile-Write\sb r)
by (cases r) auto

lemma is-non-volatile-split:
¬ is-volatile r = (is-non-volatile-Read\sb r ∨ is-non-volatile-Write\sb r ∨ is-Prog\sb r ∨ is-Ghost\sb r)
by (cases r) auto

fun outstanding-refs:: ('p memref ⇒ bool) ⇒ 'p memref list ⇒ addr set
where
outstanding-refs P [] = {}
| outstanding-refs P (r#rs) = (if P r then (address-of r) ∪ (outstanding-refs P rs)
else outstanding-refs P rs)

lemma outstanding-refs-conv: outstanding-refs P sb = ∪ (address-of ' r. r ∈ set sb ∧ P r)
by (induct sb) auto

lemma outstanding-refs-append:
⋀ ys. outstanding-refs vol (xs@ys) = outstanding-refs vol xs ∪ outstanding-refs vol ys
by (auto simp add: outstanding-refs-conv)

**lemma** outstanding-refs-empty-negate: (outstanding-refs P sb = {}) ⇒ (outstanding-refs (Not ◦ P) sb = ∪ (address-of ' set sb))
by (auto simp add: outstanding-refs-conv)

**lemma** outstanding-refs-mono-pred:
  \[ \forall r. P r \rightarrow P' r \Rightarrow \text{outstanding-refs } P \text{ sb } \subseteq \text{outstanding-refs } P' \text{ sb} \]
by (auto simp add: outstanding-refs-conv)

**lemma** outstanding-refs-mono-set:
  \[ \forall sb sb'. \text{set sb } \subseteq \text{set sb'} \Rightarrow \text{outstanding-refs } P \text{ sb } \subseteq \text{outstanding-refs } P \text{ sb'} \]
by (auto simp add: outstanding-refs-conv)

**lemma** outstanding-refs-takeWhile:
outstanding-refs P (takeWhile P' sb) \subseteq outstanding-refs P sb
apply (rule outstanding-refs-mono-set)
apply (auto dest: set-takeWhileD)
done

**lemma** outstanding-refs-subsets:
outstanding-refs is-volatile-Write sb sb \subseteq outstanding-refs is-Write sb sb
outstanding-refs is-non-volatile-Write sb sb \subseteq outstanding-refs is-Write sb sb

outstanding-refs is-volatile-Read sb sb \subseteq outstanding-refs is-Read sb sb
outstanding-refs is-non-volatile-Read sb sb \subseteq outstanding-refs is-Read sb sb

outstanding-refs is-non-volatile-Write sb sb \subseteq outstanding-refs (Not ◦ is-volatile) sb
outstanding-refs is-non-volatile-Read sb sb \subseteq outstanding-refs (Not ◦ is-volatile) sb

outstanding-refs is-volatile-Write sb sb \subseteq outstanding-refs (is-volatile) sb
outstanding-refs is-volatile-Read sb sb \subseteq outstanding-refs (is-volatile) sb

outstanding-refs is-non-volatile-Write sb sb \subseteq outstanding-refs (Not ◦ is-volatile-Write sb sb)
outstanding-refs is-non-volatile-Read sb sb \subseteq outstanding-refs (Not ◦ is-volatile-Write sb sb)

by (auto intro!: outstanding-refs-mono-pred simp add: is-volatile-Write sb sb-def is-non-volatile-Write sb sb-def is-volatile-Read sb sb-def is-non-volatile-Read sb sb-def is-Read sb sb-def split: memref.splits)

**lemma** outstanding-non-volatile-refs-conv:
outstanding-refs (Not ◦ is-volatile) sb =
outstanding-refs is-non-volatile-Write\(_{sb}\) \(\cup\) outstanding-refs is-non-volatile-Read\(_{sb}\) sb
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a, auto)
done

lemma outstanding-volatile-refs-conv:
  outstanding-refs is-volatile sb =
  outstanding-refs is-volatile-Write\(_{sb}\) sb \(\cup\) outstanding-refs is-volatile-Read\(_{sb}\) sb
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a, auto)
done

lemma outstanding-is-Write\(_{sb}\)-refs-conv:
  outstanding-refs is-Write\(_{sb}\) sb =
  outstanding-refs is-non-volatile-Write\(_{sb}\) sb \(\cup\) outstanding-refs is-volatile-Write\(_{sb}\) sb
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a, auto)
done

lemma outstanding-is-Read\(_{sb}\)-refs-conv:
  outstanding-refs is-Read\(_{sb}\) sb =
  outstanding-refs is-non-volatile-Read\(_{sb}\) sb \(\cup\) outstanding-refs is-volatile-Read\(_{sb}\) sb
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a, auto)
done

lemma outstanding-not-volatile-Read\(_{sb}\)-refs-conv: outstanding-refs (Not \(\circ\) is-volatile-Read\(_{sb}\)) sb =
  outstanding-refs is-Write\(_{sb}\) sb \(\cup\) outstanding-refs is-non-volatile-Read\(_{sb}\) sb
apply (induct sb)
apply (clarsimp)
subgoal for a sb
  by (case-tac a, auto)
done

lemmasmisc-outstanding-refs-convs = outstanding-non-volatile-refs-conv
outstanding-volatile-refs-conv
outstanding-is-Write\(_{sb}\)-refs-conv outstanding-is-Read\(_{sb}\)-refs-conv
outstanding-not-volatile-Read\(_{sb}\)-refs-conv

126
lemma no-outstanding-vol-write-takeWhile-append: outstanding-refs is-volatile-Write sb
  sb = {} \implies
  takeWhile (Not \circ is-volatile-Write sb) (sb@xs) = sb@(takeWhile (Not \circ is-volatile-Write sb) xs)
apply (induct sb)
apply (auto split: if-split-asm)
done

lemma outstanding-vol-write-takeWhile-append: outstanding-refs is-volatile-Write sb \neq
  \{} \implies
  takeWhile (Not \circ is-volatile-Write sb) (sb@xs) = (takeWhile (Not \circ is-volatile-Write sb) sb)
apply (induct sb)
apply (auto split: if-split-asm)
done

lemma no-outstanding-vol-write-dropWhile-append: outstanding-refs is-volatile-Write sb
  sb = {} \implies
  dropWhile (Not \circ is-volatile-Write sb) (sb@xs) = (dropWhile (Not \circ is-volatile-Write sb) xs)
apply (induct sb)
apply (auto split: if-split-asm)
done

lemma outstanding-vol-write-dropWhile-append: outstanding-refs is-volatile-Write sb \neq
  {} \implies
  dropWhile (Not \circ is-volatile-Write sb) (sb@xs) = (dropWhile (Not \circ is-volatile-Write sb) sb)@xs
apply (induct sb)
apply (auto split: if-split-asm)
done

lemmas outstanding-vol-write-take-drop-appends =
no-outstanding-vol-write-takeWhile-append
outstanding-vol-write-takeWhile-append
no-outstanding-vol-write-dropWhile-append
outstanding-vol-write-dropWhile-append

lemma outstanding-refs-is-non-volatile-Write sb-takeWhile-conv:
  outstanding-refs is-non-volatile-Write sb (takeWhile (Not \circ is-volatile-Write sb) sb) =
  outstanding-refs is-Write sb (takeWhile (Not \circ is-volatile-Write sb) sb)
apply (induct sb)
apply clarsimp
subgoal for a sb
  by (case-tac a, auto)
done

lemma dropWhile-not-vol-write-empty:
outstanding-refs is-volatile-Write \texttt{sb} \texttt{sb} = \{\} \implies (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write} \ \texttt{sb}) \ \texttt{sb}) = []

apply (induct \texttt{sb})
apply (auto split: if-split-asm)
done

\textbf{lemma} \text{takeWhile-not-vol-write-outstanding-refs}:
outstanding-refs is-volatile-Write \texttt{sb} (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write} \ \texttt{sb}) \ \texttt{sb}) = []
apply (induct \texttt{sb})
apply (auto split: if-split-asm)
done

\textbf{lemma} \text{no-volatile-Write} \texttt{sb} \texttt{s-conv}: (outstanding-refs is-volatile-Write \ \texttt{sb} \ \texttt{sb} = \{\}) =
(\forall r \in \text{set} \ \texttt{sb}. \ (\forall v' \ \text{sop} a' A L R W. \ r \neq \text{Write}_{\texttt{sb}} \ \text{True} a' \ \text{sop} v' A L R W))
by (force simp add: outstanding-refs-conv is-volatile-Write \texttt{sb}-def split: memref.splits)

\textbf{lemma} \text{no-volatile-Read} \texttt{sb} \texttt{s-conv}: (outstanding-refs is-volatile-Read \ \texttt{sb} \ \texttt{sb} = \{\}) =
(\forall r \in \text{set} \ \texttt{sb}. \ (\forall v' t' a'. \ r \neq \text{Read}_{\texttt{sb}} \ \text{True} a' t' v'))
by (force simp add: outstanding-refs-conv is-volatile-Read \texttt{sb}-def split: memref.splits)

\textbf{inductive} \texttt{sb-memop-step} :: (\text{instrs} \times \text{tmps} \times 'p store-buffer \times \text{memory} \times 'dirty \times 'owns \times 'rels \times 'shared) \Rightarrow
(\text{instrs} \times \text{tmps} \times 'p store-buffer \times \text{memory} \times 'dirty \times 'owns \times 'rels \times 'shared) \Rightarrow \text{bool}
(-\rightarrow_{\texttt{sb}} - [60,60] 100)
where
\textbf{SBReadBuffered}:
[\text{buffered-val} \texttt{sb} \ a = \text{Some} \ v] \implies
(\text{Read volatile} \ a \ t \ # \ is,\emptyset, \texttt{sb}, \text{m}, \text{D}, \text{O}, \text{R}, \text{S}) \rightarrow_{\texttt{sb}}
(\text{is}, \emptyset \ (t\rightarrow v), \texttt{sb}, \text{m}, \text{D}, \text{O}, \text{R}, \text{S})

\textbf{SBReadUnbuffered}:
[\text{buffered-val} \texttt{sb} \ a = \text{None}] \implies
(\text{Read volatile} \ a \ t \ # \ is, \emptyset, \texttt{sb}, \text{m}, \text{D}, \text{O}, \text{R}, \text{S}) \rightarrow_{\texttt{sb}}
(\text{is}, \emptyset \ (t\rightarrow m \ a), \texttt{sb}, \text{m}, \text{D}, \text{O}, \text{R}, \text{S})

\textbf{SBWriteNonVolatile}:
(\text{Write False} \ a \ (\text{D},f) A L R W \ # \ is, \emptyset, \texttt{sb}, \text{m}, \text{D}, \text{O}, \text{R}, \text{S}) \rightarrow_{\texttt{sb}}
(\text{is}, \emptyset, \texttt{sb}@[\text{Write}_{\texttt{sb}} \ \text{False} \ a \ (\text{D},f) \ (f \ \emptyset) A L R W], \text{m}, \text{D}, \text{O}, \text{R}, \text{S})

\textbf{SBWriteVolatile}:
(\text{Write True} \ a \ (\text{D},f) A L R W \ # \ is, \emptyset, \texttt{sb}, \text{m}, \text{D}, \text{O}, \text{R}, \text{S}) \rightarrow_{\texttt{sb}}
(\text{is}, \emptyset, \texttt{sb}@[\text{Write}_{\texttt{sb}} \ \text{True} \ a \ (\text{D},f) \ (f \ \emptyset) A L R W], \text{m}, \text{D}, \text{O}, \text{R}, \text{S})

\textbf{SBFence}:
(Fence # is, \(\emptyset\), [], m, \(D, O, R, S\)) \(\rightarrow_{sb}\) (is, \(\emptyset\), [], m, \(D, O, R, S\))

<table>
<thead>
<tr>
<th>SBRMWRReadOnly:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\neg \text{cond } (\emptyset(t\mapsto m a))] (\implies)</td>
</tr>
<tr>
<td>(RMW a t (D,f) cond ret A L R W# is, (\emptyset), [], m, (D, O, R, S)) (\rightarrow_{sb}) (is, (\emptyset(t\mapsto m a)), [], m, (D, O, R, S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBRMWRWrite:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{cond } (\emptyset(t\mapsto m a))] (\implies)</td>
</tr>
<tr>
<td>(RMW a t (D,f) cond ret A L R W# is, (\emptyset), [], m, (D, O, R, S)) (\rightarrow_{sb})</td>
</tr>
<tr>
<td>(is, (\emptyset(t\mapsto \text{ret } (m a))), [], m(a:= f(\emptyset(t\mapsto m a))), D, O, R, S)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBGhost:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ghost A L R W# is, (\emptyset), sb, m, D, O, R, S) (\rightarrow_{sb})</td>
</tr>
</tbody>
</table>

\textbf{inductive} sbh-memop-step ::
\[
\begin{align*}
\text{instrs} \times \text{tmps} \times \text{\textquote{p store-buffer}} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{rels} \times \text{shared} \\
\end{align*}
\]
\(\Rightarrow\)
\[
\begin{align*}
\text{instrs} \times \text{tmps} \times \text{\textquote{p store-buffer}} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{rels} \times \text{shared} \\
\end{align*}
\(\Rightarrow\) bool
\[
\begin{align*}
(\neg \rightarrow_{sb} - [60,60] 100) \\
\end{align*}
\]
\textbf{where}

<table>
<thead>
<tr>
<th>SBHReadBuffered:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{buffered-val sb a = Some } v] (\implies)</td>
</tr>
<tr>
<td>(Read volatile a t # is, (\emptyset), sb, m, (D, O, R, S)) (\rightarrow_{sb})</td>
</tr>
<tr>
<td>(is, (\emptyset) (t(\mapsto v)), sb@[\text{Read}_{sb} \text{volatile a t v}], m, (D, O, R, S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBHReadUnbuffered:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{buffered-val sb a = None}] (\implies)</td>
</tr>
<tr>
<td>(Read volatile a t # is, (\emptyset), sb, m, (D, O, R, S)) (\rightarrow_{sb})</td>
</tr>
<tr>
<td>(is, (\emptyset) (t(\mapsto m a)), sb@[\text{Read}_{sb} \text{volatile a t (m a)]}, m, (D, O, R, S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBHWriteNonVolatile:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Write False a (D,f) A L R W#is, (\emptyset), sb, m, (D, O, R, S)) (\rightarrow_{sb})</td>
</tr>
<tr>
<td>(is, (\emptyset), sb@[\text{Write}_{sb} \text{False a (D,f) (f } \emptyset}) A L R W], m, (D, O, R, S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBHWriteVolatile:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Write True a (D,f) A L R W# is, (\emptyset), sb, m, (D, O, R, S)) (\rightarrow_{sb})</td>
</tr>
<tr>
<td>(is, (\emptyset), sb@[\text{Write}_{sb} \text{True a (D,f) (f } \emptyset}) A L R W], m, True, (O, R, S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBHFence:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Fence # is, (\emptyset), [], m, (D, O, R, S)) (\rightarrow_{sb}) (is, (\emptyset), [], m, False, (O, \text{Map.empty}, S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBHRMWReadOnly:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\neg \text{cond } (\emptyset(t\mapsto m a))] (\implies)</td>
</tr>
</tbody>
</table>

129
(RMW a t (D,f) cond ret A L R W# is, \( \emptyset \), [], m, \( D, O, R, S \)) \( \stackrel{sbh}{\rightarrow} \) (is, \( \emptyset(t\mapsto m \ a) \), [], m, False, \( O, Map.\emptyset, S \))

<table>
<thead>
<tr>
<th>SBHRMWWrite:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cond ( \emptyset(t\mapsto m \ a) ) ( \Rightarrow ) (RMW a t (D,f) cond ret A L R W# is, ( \emptyset ), [], m, ( D, O, R, S )) ( \rightarrow_{sbh} ) (is, ( \emptyset(t\mapsto ret (m \ a)) ), [], m(a:= f(( \emptyset(t\mapsto m \ a) ))), False, ( O \cup A - R, Map.\emptyset, S \oplus W R \ominus A L ))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBHGhost:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ghost A L R W# is, ( \emptyset ), sb, m, ( D, O, R, S )) ( \rightarrow_{sbh} ) (is, ( \emptyset ), sb[( \text{ Ghost } sb ) A L R W], m, ( D, O, R, S ))</td>
</tr>
</tbody>
</table>

**interpretation** direct: memory-system direct-memop-step id-storebuffer-step .


**primrec** non-volatile-owned-or-read-only:: bool \( \Rightarrow \) shared \( \Rightarrow \) owns \( \Rightarrow \) 'a memref list \( \Rightarrow \) bool

**where**

non-volatile-owned-or-read-only pending-write \( \emptyset \) \( \Rightarrow \) True

<table>
<thead>
<tr>
<th>non-volatile-owned-or-read-only pending-write ( \emptyset ) ( x#xs ) =</th>
</tr>
</thead>
<tbody>
<tr>
<td>(case x of</td>
</tr>
</tbody>
</table>

Read_{sb} volatile a t v \( \Rightarrow \)

(\( \neg \text{volatile} \rightarrow \text{pending-write} \rightarrow (a \in O \lor a \in \text{read-only } S) \)) \( \land \)

non-volatile-owned-or-read-only pending-write \( \emptyset \) xs

| Write_{sb} volatile a sop v A L R W \( \Rightarrow \)

(if volatile then non-volatile-owned-or-read-only True \( S \oplus W R \ominus A L \) \( (O \cup A - R) \) xs

else a \( \in O \land \) non-volatile-owned-or-read-only pending-write \( \emptyset \) xs)

| Ghost_{sb} A L R W \( \Rightarrow \) non-volatile-owned-or-read-only pending-write \( S \oplus W R \ominus A L \)

\( (O \cup A - R) \) xs

| - \( \Rightarrow \) non-volatile-owned-or-read-only pending-write \( \emptyset \) xs)

**primrec** acquired :: bool \( \Rightarrow \) 'a memref list \( \Rightarrow \) addr set \( \Rightarrow \) addr set

**where**

acquired pending-write \( \emptyset \) A = (if pending-write then A else \( \{} \))

<table>
<thead>
<tr>
<th>acquired pending-write ( x#xs ) A =</th>
</tr>
</thead>
<tbody>
<tr>
<td>(case x of</td>
</tr>
</tbody>
</table>

Write_{sb} volatile - - - A’ L R W \( \Rightarrow \)

(if volatile then acquired True xs (if pending-write then \( A \cup A’ - R \) else \( A’ - R) \))

else acquired pending-write xs A)

| Ghost_{sb} A’ L R W \( \Rightarrow \) acquired pending-write xs (if pending-write then \( A \cup A’ - R \) else A)

| - \( \Rightarrow \) acquired pending-write xs A)

**primrec** share :: 'a memref list \( \Rightarrow \) shared \( \Rightarrow \) shared

**where**
share [] S = S
| share (x#xs) S =
  (case x of
    Write sb volatile - - - A L R W ⇒ (if volatile then (share xs (S ⊕ W R ⊕A L)) else share xs S)
  | Ghost sb A L R W ⇒ share xs (S ⊕ W R ⊕A L)
  | - ⇒ share xs S)

primrec acquired-reads :: bool ⇒ 'a memref list ⇒ addr set ⇒ addr set
where
acquired-reads pending-write [] A = {}
| acquired-reads pending-write (x#xs) A =
  (case x of
    Read sb volatile a t v ⇒ (if pending-write ∧ ¬ volatile ∧ a ∈ A
                                then insert a (acquired-reads pending-write xs A)
                                else acquired-reads pending-write xs A)
    | Write sb volatile - - - A′ L R W ⇒
      (if volatile then acquired-reads True xs (if pending-write then (A ∪ A′ − R) else (A′ − R))
       else acquired-reads pending-write xs A)
    | Ghost sb A′ L R W ⇒ acquired-reads pending-write xs (A ∪ A′ − R)
    | - ⇒ acquired-reads pending-write xs A)

lemma union-mono-aux: A ⊆ B ⇒ A ∪ C ⊆ B ∪ C
  by blast

lemma set-minus-mono-aux: A ⊆ B ⇒ A − C ⊆ B − C
  by blast

lemma acquired-mono: \( \bigwedge A B \) pending-write. A ⊆ B ⇒ acquired pending-write xs A ⊆ acquired pending-write xs B
apply (induct xs)
apply simp
subgoal for a xs A B pending-write
apply (case-tac a )
apply clarsimp
subgoal for volatile a1 D f v A′ L R W x
  apply (drule-tac C=A′ in union-mono-aux)
  apply (drule-tac C=R in set-minus-mono-aux)
  apply blast
done
apply clarsimp
apply clarsimp
apply clarsimp
subgoal for A′ L R W x
  apply (drule-tac C=A′ in union-mono-aux)
  apply (drule-tac C=R in set-minus-mono-aux)
  apply blast
done
done
done

lemma acquired-mono-in:
  assumes x-in: $x \in \text{acquired pending-write} \, \text{xs A}$
  assumes sub: $A \subseteq B$
  shows $x \in \text{acquired pending-write} \, \text{xs B}$
using acquired-mono [OF sub, of pending-write xs] x-in
by blast

lemma acquired-no-pending-write:\( \forall A \, B. \, \text{acquired False} \, \text{xs A} = \text{acquired False} \, \text{xs B} \)
by (induct xs) (auto split: memref.splits)

lemma acquired-no-pending-write-in:
  $x \in \text{acquired False} \, \text{xs A} \implies x \in \text{acquired False} \, \text{xs B}$
apply (subst acquired-no-pending-write)
apply auto
done

lemma acquired-pending-write-mono-in:\( \forall A \, B. \, x \in \text{acquired False} \, \text{xs A} \implies x \in \text{acquired True} \, \text{xs B} \)
apply (induct xs)
apply (auto split: memref.splits if-split-asm intro: acquired-mono-in)
done

lemma acquired-pending-write-mono: acquired False xs A \subseteq acquired True xs B
by (auto intro: acquired-pending-write-mono-in)

lemma acquired-append:\( \forall A \, \text{pending-write} \, \text{acquired pending-write} \, (\text{xs@ys}) \, A = \text{acquired} \, (\text{pending-write} \lor \text{outstanding-refs is-volatile-Write}_{sb} \, \text{xs} \neq \{\}) \, \text{ys} \, (\text{acquired pending-write} \, \text{xs A}) \)
apply (induct xs)
apply (auto split: memref.splits intro: acquired-no-pending-write-in)
done

lemma acquired-take-drop:
acquired (pending-write \lor \text{outstanding-refs is-volatile-Write}_{sb} \, (\text{takeWhile P xs}) \neq \{\})
(dropWhile P xs) (acquired pending-write (takeWhile P xs) A) =
acquired pending-write xs A
proof
  have acquired pending-write xs A = acquired pending-write ((\text{takeWhile P xs})@(\text{dropWhile P xs})) A
  by simp
also
  from acquired-append [where xs=(\text{takeWhile P xs}) and ys=(\text{dropWhile P xs})]
  have \ldots = acquired (pending-write \lor \text{outstanding-refs is-volatile-Write}_{sb} \, (\text{takeWhile P xs}) \neq \{\})
  (dropWhile P xs) (acquired pending-write (takeWhile P xs) A)
  by simp
finally show \?thesis
by simp

qed

**lemma** share-mono: \( \forall A. B. \text{dom } A \subseteq \text{dom } B \implies \text{dom } (\text{share } xs \ A) \subseteq \text{dom } (\text{share } xs \ B) \)

apply (induct xs)

apply simp

subgoal for a xs A B

apply (case-tac a)

apply (clarsimp iff del: domIff)

subgoal for volatile a1 D f v A’ L R W x

apply (drule-tac C=R and x=W in augment-mono-aux)

apply (drule-tac C=L in restrict-mono-aux)

apply blast

done

apply clarsimp

apply clarsimp

apply (clarsimp iff del: domIff)

subgoal for A’ L R W x

apply (drule-tac C=R and x=W in augment-mono-aux)

apply (drule-tac C=L in restrict-mono-aux)

apply blast

done

done

done

**lemma** share-mono-in:

assumes x-in: \( x \in \text{dom } (\text{share } xs \ A) \)

assumes sub: \( \text{dom } A \subseteq \text{dom } B \)

shows \( x \in \text{dom } (\text{share } xs \ B) \)

using share-mono [OF sub, of xs] x-in

by blast

**lemma** acquired-reads-mono:

\( \forall A. B. \text{pending-write. } A \subseteq B \implies \text{acquired-reads } \text{pending-write } xs \ A \subseteq \text{acquired-reads } \text{pending-write } xs \ B \)

apply (induct xs)

apply simp

subgoal for a xs A B pending-write

apply (case-tac a)

apply clarsimp

subgoal for volatile a1 D f v A’ L R W x

apply (drule-tac C=A’ in union-mono-aux)

apply (drule-tac C=R in set-minus-mono-aux)

apply blast

done

apply clarsimp

apply blast

apply clarsimp

apply clarsimp

subgoal for A’ L R W x

133
apply (drule-tac C=A' in union-mono-aux)
apply (drule-tac C=R in set-minus-mono-aux)
apply blast
done
done
done

lemma acquired-reads-mono-in:
  assumes x-in: x ∈ acquired-reads pending-write xs A
  assumes sub: A ⊆ B
  shows x ∈ acquired-reads pending-write xs B
using acquired-reads-mono [OF sub, of pending-write xs] x-in
by blast

lemma acquired-reads-no-pending-write: ∀A B. acquired-reads False xs A =
acquired-reads False xs B
by (induct xs) (auto split: memref.splits)

lemma acquired-reads-no-pending-write-in:
x ∈ acquired-reads False xs A → x ∈ acquired-reads False xs B
apply (subst acquired-reads-no-pending-write)
apply blast
done

lemma acquired-reads-pending-write-mono:
  ∀A. acquired-reads False xs A ⊆ acquired-reads True xs A
by (induct xs) (auto split: memref.splits intro: acquired-reads-mono-in )

lemma acquired-reads-pending-write-mono-in:
  assumes x-in: x ∈ acquired-reads False xs A
  shows x ∈ acquired-reads True xs A
using acquired-reads-pending-write-mono [of xs A] x-in
by blast

lemma acquired-reads-append: ∀pending-write A. acquired-reads pending-write (xs@ys) A =
acquired-reads pending-write xs A ∪
acquired-reads (pending-write ∨ (outstanding-refs is-volatile-Write a sb xs ≠ {}) ) ys
(acquired pending-write xs A)
proof (induct xs)
case Nil thus ?case by (auto dest: acquired-reads-no-pending-write-in)
next
case (Cons x xs)
show ?case
proof (cases x)
case (Write a sb volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case False
show ?thesis
using Cons.hyps
by (auto simp add: WriteFalse)
next
case True
show ?thesis
using Cons.hyps
by (auto simp add: WriteTrue)
qed
next
case (ReadFalse volatile a t v)
show ?thesis
proof (cases volatile)
case False
show ?thesis
using Cons.hyps
by (auto simp add: ReadFalse)
next
case True
show ?thesis
using Cons.hyps
by (auto simp add: ReadTrue)
qed
next
case ProgFalse
with Cons.hyps show ?thesis by auto
next
case (GhostFalse A' L R W)
have (acquired False xs (A ∪ A' − R )) = (acquired False xs A)
by (simp add: acquired-no-pending-write)
with Cons.hyps show ?thesis by (auto simp add: GhostFalse)
qed
qed

lemma in-acquired-reads-no-pending-write-outstanding-write:
\( A. a \in \text{acquired-reads False xs A } \implies \text{outstanding-refs (is-volatile-WriteFalse) xs } \neq \{ \} \)
apply (induct xs)
apply simp
apply (auto split: memref.splits)
apply auto
done

lemma augment-read-only-mono: read-only \( S \subseteq \text{read-only } S' \implies \)
read-only \( (S \oplus W R) \subseteq \text{read-only } (S' \oplus W R) \)
by (auto simp add: augment-shared-def read-only-def)

lemma restrict-read-only-mono: read-only \( S \subseteq \text{read-only } S' \implies \)
read-only \( (S \ominus A L) \subseteq \text{read-only } (S' \ominus A L) \)
apply (clarsimp simp add: restrict-shared-def read-only-def split: option.splits if-split-asm)
apply (rule conjI)
apply blast
apply fastforce
done

lemma share-read-only-mono: \( \forall S, S'. \text{ read-only } S \subseteq \text{ read-only } S' \implies \text{ read-only } (\text{share sb } S) \subseteq \text{ read-only } (\text{share sb } S') \)

proof (induct sb)
  case Nil thus ?case by simp

next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case False
      with Cons Write sb show ?thesis
      by auto
    next
      case True
      note \( (\text{read-only } S \subseteq \text{ read-only } S') \)
      from augment-read-only-mono [OF this]
      have \( \text{read-only } (S \oplus W R) \subseteq \text{ read-only } (S' \oplus W R) \).
      from restrict-read-only-mono [OF this, of A L]
      have \( \text{read-only } (S \oplus W R \ominus A L) \subseteq \text{ read-only } (S' \oplus W R \ominus A L) \).
      from Cons.hyps [OF this]
      show ?thesis
      by (clarsimp simp add: Write sb True)
  qed

next
  case Read sb with Cons show ?thesis
  by auto

next
  case Prog sb with Cons show ?thesis
  by auto

next
  case (Ghost sb A L R W)
  note \( (\text{read-only } S \subseteq \text{ read-only } S') \)
  from augment-read-only-mono [OF this]
  have \( \text{read-only } (S \oplus W R) \subseteq \text{ read-only } (S' \oplus W R) \).
  from restrict-read-only-mono [OF this, of A L]
  have \( \text{read-only } (S \oplus W R \ominus A L) \subseteq \text{ read-only } (S' \oplus W R \ominus A L) \).
  from Cons.hyps [OF this]
  show ?thesis
  by (clarsimp simp add: Ghost sb)
qed

qed
lemma non-volatile-owned-or-read-only-append: 
\( \forall O S \) pending-write. non-volatile-owned-or-read-only pending-write \( S O (xs@ys) \) 
= (non-volatile-owned-or-read-only pending-write \( S O xs \) \( \land \) 
non-volatile-owned-or-read-only (pending-write \( \lor \) outstanding-refs is-volatile-Write sb xs \( \neq \) \{\})) 
(share xs S) (acquired True xs O) ys 
apply (induct xs) 
apply (auto split: memref.splits) 
done

lemma non-volatile-owned-or-read-only-mono: 
\( \forall O O' S \) pending-write. \( O \subseteq O' \implies \) non-volatile-owned-or-read-only pending-write \( S O \) 
xs 
\implies \) non-volatile-owned-or-read-only pending-write \( S O' xs \) 
apply (induct xs) 
apply simp 
subgoal for a xs O O' S pending-write 
apply (case-tac a) 
apply (clarsimp split: if-split-asm) 
subgoal for volatile a1 D f v A L R W 
apply (drule-tac C=A in union-mono-aux) 
apply (drule-tac C=R in set-minus-mono-aux) 
apply blast 
done 
apply fastforce 
apply fastforce 
apply fastforce 
apply clarsimp 
subgoal for A L R W 
apply (drule-tac C=A in union-mono-aux) 
apply (drule-tac C=R in set-minus-mono-aux) 
apply blast 
done 
done 
done

lemma non-volatile-owned-or-read-only-shared-mono: 
\( \forall S S' O \) pending-write. \( S \subseteq S' \implies \) non-volatile-owned-or-read-only pending-write \( S O \) 
xs 
\implies \) non-volatile-owned-or-read-only pending-write \( S' O xs \) 
apply (induct xs) 
apply simp 
subgoal for a xs S S' O pending-write 
apply (case-tac a) 
apply (clarsimp split: if-split-asm) 
subgoal for volatile a1 D f v A L R W 
apply (frule-tac C=R and x=W in augment-mono-map) 
apply (drule-tac A=S c=W R and C=L in restrict-mono-map) 
apply (fastforce dest: read-only-mono) 
done
apply (fastforce dest: read-only-mono shared-leD)
apply fastforce
subgoal for A L R W
apply (frule-tac C=R and x=W in augment-mono-map)
apply (drule-tac A=S ⊕ W R and C=L in restrict-mono-map)
apply (fastforce dest: read-only-mono)
done
done
done

lemma non-volatile-owned-or-read-only-pending-write-antimono:
\( \forall O. \text{non-volatile-owned-or-read-only True } S \ O \ xs \implies \text{non-volatile-owned-or-read-only False } S \ O \ xs \)
by (induct xs) (auto split: memref.splits)

primrec all-acquired :: 'a memref list ⇒ addr set
where
  all-acquired [] = {}
| all-acquired (i#is) =
  (case i of
    Write sb volatile - - - A L R W ⇒ (if volatile then A ∪ all-acquired is else all-acquired is)
    | Ghost sb A L R W ⇒ A ∪ all-acquired is
    | - ⇒ all-acquired is)

lemma all-acquired-append: all-acquired (xs@ys) = all-acquired xs ∪ all-acquired ys
apply (induct xs)
apply (auto split: memref.splits)
done

lemma acquired-reads-all-acquired: \( \forall O \text{ pending-write. acquired-reads pending-write sb } O \subseteq O \cup \text{all-acquired sb} \)
apply (induct sb)
apply clarsimp
apply (auto split: memref.splits)
done

lemma acquired-takeWhile-non-volatile-Write sb:
\( \forall A. (\text{acquired True (takeWhile (Not o is-volatile-Write sb) sb) A}) \subseteq A \cup \text{all-acquired (takeWhile (Not o is-volatile-Write sb) sb)} \)
apply (induct sb)
apply clarsimp
subgoal for a sb A
apply (case-tac a)
apply auto
done
done

done

lemma acquired-False-takeWhile-non-volatile-Write sb:
acquired False (takeWhile (Not ◦ is-volatile-Writeₐᵇ) sb) A = {}
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a) auto
done

lemma outstanding-refs-takeWhile-opposite: outstanding-refs P (takeWhile (Not ◦ P) xs) = {}  
apply (induct xs)
apply auto
done

lemma no-outstanding-volatile-Writeₐᵇ-acquired:  
outstanding-refs is-volatile-Writeₐᵇ sb = {} implies acquired False sb A = {}
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a) auto
done

lemma acquired-all-acquired: ⋀ pending-write A. acquired pending-write xs A ⊆ A ∪ all-acquired xs  
apply (induct xs)
apply (auto split: memref.splits)
done

lemma acquired-all-acquired-in: x ∈ acquired pending-write xs A ⇒ x ∈ A ∪ all-acquired xs  
using acquired-all-acquired
by blast

primrec sharing-consistent:: shared ⇒ owns ⇒ 'a memref list ⇒ bool
where
  sharing-consistent S O [] = True
| sharing-consistent S O (r#rs) =  
  (case r of
    Writeₐᵇ volatile - - A L R W ⇒  
    (if volatile then A ⊆ dom S ∪ O ∧ L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ O ∧  
      sharing-consistent (S ⊕ₘ W R ⊖ₐ L) (O ∪ A − R) rs
    else sharing-consistent S O rs)
  | Ghostₐᵇ A L R W ⇒ A ⊆ dom S ∪ O ∧ L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ O ∧  
    sharing-consistent (S ⊕ₘ W R ⊖ₐ L) (O ∪ A − R) rs
  | - ⇒ sharing-consistent S O rs)

lemma sharing-consistent-all-acquired:  
⋀ S O. sharing-consistent S O sb ⇒ all-acquired sb ⊆ dom S ∪ O
proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
proof (cases x)
  case (Write\textsubscript{sb} volatile a sop v A L R W)
  show ?thesis
  proof (cases volatile)
  case False
  with Cons Write\textsubscript{sb} show ?thesis by auto
  next
  case True
  from Cons.hyps [where \( S=(S \oplus W R \ominus A) \) and \( O=(O \cup A \ominus R) \)] Cons.prems
  show ?thesis
  by (auto simp add: Write\textsubscript{sb} True)
  qed
  qed

lemma sharing-consistent-append:
\( \bigwedge \{ S, O \} \text{ sharing-consistent } S O \ (xs@ys) = \)
\[ (\text{sharing-consistent } S O \ \text{xs} \land \text{sharing-consistent } (\text{share } xs S) \ \text{(acquired } \text{True } \text{xs } O) \ \text{ys}) \]
apply (induct xs)
apply (auto split: memref.splits)
done

primrec read-only-reads :: owns \Rightarrow \'a memref list \Rightarrow addr set
where
read-only-reads \( O \ [] = \{ \} \)
| read-only-reads \( O \ (x#xs) = \)
  (case x of
    Read\textsubscript{sb} volatile a t v \Rightarrow (if \neg \text{volatile} \land a \notin O
    then insert a (read-only-reads O xs)
    else read-only-reads O xs)
  | Write\textsubscript{sb} volatile - - - A L R W \Rightarrow
    (if volatile then read-only-reads (O \cup A \ominus R) \text{xs}
    else read-only-reads O \text{xs} )
  | Ghost\textsubscript{sb} A L R W \Rightarrow read-only-reads (O \cup A \ominus R) \text{xs}
  | - \Rightarrow \text{read-only-reads } O \text{xs} )
  140
lemma read-only-reads-append:
\( \bigwedge O. \) read-only-reads \( O \) (xs@ys) =
read-only-reads \( O \) xs \( \cup \) read-only-reads (acquired True xs \( O \)) ys
  apply (induct xs)
  apply simp
subgoal for a xs \( O \)
    by (case-tac a) auto
done

lemma read-only-reads-antimono:
\( \bigwedge O O'. \)
\( O \subseteq O' \Rightarrow \) read-only-reads \( O' \) sb \( \subseteq \) read-only-reads \( O \) sb
  apply (induct sb)
  apply simp
subgoal for a sb \( O O' \)
  apply (case-tac a)
  apply (clarsimp split: if-split-asm)
    subgoal for volatile a1 D f v A L R W
      apply (drule-tac C=A in union-mono-aux)
      apply (drule-tac C=R in set-minus-mono-aux)
      apply blast
done
  apply auto
subgoal for A L R W x
  apply (drule-tac C=A in union-mono-aux)
  apply (drule-tac C=R in set-minus-mono-aux)
  apply blast
done
done
done

primrec non-volatile-writes-unshared:: shared \( \Rightarrow \) 'a memref list \( \Rightarrow \) bool
where
  non-volatile-writes-unshared \( S [] \) = True
| non-volatile-writes-unshared \( S (x#xs) \) =
  (case x of
    Write\( sb \) volatile a1 D f v A L R W \( \Rightarrow \) (if volatile then non-volatile-writes-unshared \( S \oplus W R \ominus A L \) xs
      else a \notin dom \( S \) \& non-volatile-writes-unshared \( S \) xs)
    | Ghost\( sb \) A L R W \( \Rightarrow \) non-volatile-writes-unshared \( S \oplus W R \ominus A L \) xs
    | - \( \Rightarrow \) non-volatile-writes-unshared \( S \) xs)

lemma non-volatile-writes-unshared-append:
\( \bigwedge S. \) non-volatile-writes-unshared \( S \) (xs@ys)
  = (non-volatile-writes-unshared \( S \) xs \( \wedge \) non-volatile-writes-unshared (share xs \( S \)) ys)
  apply (induct xs)
  apply (auto split: memref.splits)
done
lemma non-volatile-writes-unshared-antimono:
\( \land S S' \). dom \( S \subseteq \) dom \( S' \) \( \implies \) non-volatile-writes-unshared \( S \) \( xs \)
\implies non-volatile-writes-unshared \( S' \) \( xs \)
apply (induct \( xs \))
apply simp
subgoal for a \( xs \) \( S S' \)
apply (case-tac a)
apply (clarsimp split: if-split-asm)
subgoal for volatile a1 D f v A L R W
apply (drule-tac C=R in augment-mono-aux)
apply (drule-tac C=L in restrict-mono-aux)
apply blast
done
apply fastforce
apply fastforce
apply fastforce
apply (clarsimp split: if-split-asm)
subgoal for A L R W
apply (drule-tac C=R in augment-mono-aux)
apply (drule-tac C=L in restrict-mono-aux)
apply blast
done
done
done
done

primrec no-write-to-read-only-memory:: shared \( \Rightarrow \) \( 'a \) memref list \( \Rightarrow \) bool
where
no-write-to-read-only-memory \( S \) [] = True
| no-write-to-read-only-memory \( S \) (x#xs) =
  (case x of
    Write\( \_sb \) volatile a sop v A L R W \( \Rightarrow \) a \( \notin \) read-only \( S \) \( \land \)
      (if volatile then no-write-to-read-only-memory \( (S \oplus_w R \ominus_A L) \) \( xs \)
else no-write-to-read-only-memory \( S \) \( xs \))
| Ghost\( \_sb \) A L R W \( \Rightarrow \) no-write-to-read-only-memory \( (S \oplus_w R \ominus_A L) \) \( xs \)
| - \( \Rightarrow \) no-write-to-read-only-memory \( S \) \( xs \))

lemma no-write-to-read-only-memory-append:
\( \land S \). no-write-to-read-only-memory \( S \) \( (xs@ys) \)
\( = \) (no-write-to-read-only-memory \( S \) \( xs \) \( \land \) no-write-to-read-only-memory (share \( xs \) \( S \)) \( ys \))
apply (induct \( xs \))
apply simp
subgoal for a \( xs \) \( S \)
  by (case-tac a) auto
done

lemma no-write-to-read-only-memory-antimono:
\( \land S S' \). \( S \subseteq_S S' \) \( \implies \) no-write-to-read-only-memory \( S' \) \( xs \)
apply (induct xs)
apply simp
subgoal for a xs S S'
apply (case-tac a)
apply (clarsimp split: if-split-asm)
subgoal for volatile a1 D f v A L R
apply (frule-tac C=R and x=W in augment-mono-map)
apply (drule-tac A=S ⊕ W and C=L and x=A in restrict-mono-map)
done
apply (fastforce dest: read-only-mono shared-leD)
done
apply (fastforce)
done
apply (clarsimp)
subgoal for A L R
apply (frule-tac C=R and x=W in augment-mono-map)
apply (drule-tac A=S ⊕ W and C=L and x=A in restrict-mono-map)
done
done
done

locale outstanding-non-volatile-refs-owned-or-read-only =
fixes S::shared
fixes ts::(′p,′p store-buffer, bool, owns, rels) thread-config list
assumes outstanding-non-volatile-refs-owned-or-read-only: ⋀ i is ORD θ sb p.
[i < length ts; ts!i = (p,i,sb, D, O, R)]
⇒
non-volatile-owned-or-read-only False S O sb

locale outstanding-volatile-writes-unowned-by-others =
fixes ts::(′p,′p store-buffer, bool, owns, rels) thread-config list
assumes outstanding-volatile-writes-unowned-by-others: ⋀ i p
[i < length ts; j < length ts; i≠j; ts!i = (p,i,sb, D, O, R); ts!j = (p,j,sb, D, O, R)]
⇒
(O_j ∪ all-acquired sb_j) ∩ outstanding-refs is-volatile-Write sb_i = {}

locale read-only-reads-unowned =
fixes ts::(′p,′p store-buffer, bool, owns, rels) thread-config list
assumes read-only-reads-unowned: ⋀ i p
[i < length ts; j < length ts; i≠j; ts!i = (p,i,sb, D, O, R); ts!j = (p,j,sb, D, O, R)]

\[ (O_i \cup \text{all-acquired } sb_j) \cap \]
\[ \text{read-only-reads (acquired True} \]
\[ (\text{takeWhile (Not } o \text{-volatile-Write}_{sb} \text{) } sb_i) \]
\[ (\text{dropWhile (Not } o \text{-volatile-Write}_{sb} \text{) } sb_j) = \{\} \]

**locale** ownership-distinct =

**fixes ts::(p, p store-buffer, bool, owns, rels) thread-config list**

**assumes** ownership-distinct:
\[ \forall i \ j \ p \ i = O_i \ R_i \ D_i \ \theta_i \ sb_i \ p_j = O_j \ R_j \ D_j \ \theta_j \ sb_j \]
\[ [i < \text{length ts}; j < \text{length ts}; i \neq j] \]
\[ \text{tsli = (p_i, is_i, sb_i, D_i, O_i, R_i); tsli} = (p_j, is_j, sb_j, D_j, O_j, R_j) \]
\[ \implies (O_i \cup \text{all-acquired } sb_j) \cap (O_j \cup \text{all-acquired } sb_j) = \{\} \]

**locale** valid-ownership =

outstanding-non-volatile-refs-owned-or-read-only +

outstanding-volatile-writes-unowned-by-others +

read-only-reads-unowned +

ownership-distinct

**locale** outstanding-non-volatile-writes-unshared =

**fixes S::shared and ts::(p, p store-buffer, bool, owns, rels) thread-config list**

**assumes** outstanding-non-volatile-writes-unshared:
\[ \forall i \ p \ i = O \ R \ D \ \theta \ sb \]
\[ [i < \text{length ts}; \text{tsli = (p, is, sb, D, O, R)}] \]
\[ \implies \non-volatile-writes-unshared S \ sb \]

**locale** sharing-consis =

**fixes S::shared and ts::(p, p store-buffer, bool, owns, rels) thread-config list**

**assumes** sharing-consis:
\[ \forall i \ p \ i = O \ R \ D \ \theta \ sb \]
\[ [i < \text{length ts}; \text{tsli = (p, is, sb, D, O, R)}] \]
\[ \implies \text{sharing-consistent } S \ O \ sb \]

**locale** no-outstanding-write-to-read-only-memory =

**fixes S::shared and ts::(p, p store-buffer, bool, owns, rels) thread-config list**

**assumes** no-outstanding-write-to-read-only-memory:
\[ \forall i \ p \ i = O \ R \ D \ \theta \ sb \]
\[ [i < \text{length ts}; \text{tsli = (p, is, sb, D, O, R)}] \]
\[ \implies \no-write-to-read-only-memory S \ sb \]

144
locale valid-sharing =
  outstanding-non-volatile-writes-unshared +
  sharing-consis +
  read-only-unowned +
  unowned-shared +
  no-outstanding-write-to-read-only-memory

locale valid-ownership-and-sharing = valid-ownership +
  outstanding-non-volatile-writes-unshared +
  sharing-consis +
  no-outstanding-write-to-read-only-memory

lemma (in read-only-reads-unowned)
read-only-reads-unowned-nth-update:
\[\forall i \ p \ is \ O \ R \ D \ q \ sb.\]
\[i < \text{length } ts; ts[i] = (p, is, q, sb, D, O, R);\]
  read-only-reads (acquired True (takeWhile (Not is-volatile-Write sb) sb') O')
  (dropWhile (Not is-volatile-Write sb) sb') \subseteq read-only-reads (acquired True
  (takeWhile (Not is-volatile-Write sb) sb) O)
  (dropWhile (Not is-volatile-Write sb) sb);

\[O' \cup \text{all-acquired } sb' \subseteq O \cup \text{all-acquired } sb\] \implies
  read-only-reads-unowned (ts[i := (p', is', q', sb', D', O', R')])

apply (unfold-locales)
apply (clarsimp simp add: nth-list-update split: if-split-asm)
apply (fastforce dest: read-only-reads-unowned)+
done

lemma outstanding-non-volatile-refs-owned-or-read-only-tl:
  outstanding-non-volatile-refs-owned-or-read-only S (t#ts) \implies

apply (clarsimp simp add: outstanding-non-volatile-refs-owned-or-read-only-def)

lemma outstanding-volatile-writes-unowned-by-others-tl:
  outstanding-volatile-writes-unowned-by-others (t#ts) \implies

apply (clarsimp simp add: outstanding-volatile-writes-unowned-by-others-def)
apply fastforce
done

lemma read-only-reads-unowned-tl:
  read-only-reads-unowned (t # ts) \implies

apply (clarsimp simp add: read-only-reads-unowned-def)
apply fastforce
done
lemma ownership-distinct-tl:
assumes dist: ownership-distinct (t#ts)
shows ownership-distinct ts
proof –
from dist
interpret ownership-distinct t#ts .

show ?thesis
proof (rule ownership-distinct.intro)
fix i j p is O R D xs sb p' is' O' R' D' xs' sb'
assume i-bound: i < length ts
and j-bound: j < length ts
and neq: i ≠ j
and ith: ts ! i = (p,is,xs,sb,D,O,R)
and jth: ts ! j = (p',is',xs',sb',D',O',R')
from i-bound j-bound neq ith jth
show (O ∪ all-acquired sb) ∩ (O' ∪ all-acquired sb') = {}
by – (rule ownership-distinct [of Suc i Suc j],auto)
qed
qed

lemma valid-ownership-tl: valid-ownership S (t#ts) ⇒ valid-ownership S ts
by (auto simp add: valid-ownership-def
intro: outstanding-volatile-writes-unowned-by-others-tl
outstanding-non-volatile-refs-owned-or-read-only-tl ownership-distinct-tl
read-only-reads-unowned-tl)

lemma sharing-consistent-takeWhile:
assumes consis: sharing-consistent S O sb
shows sharing-consistent S O (takeWhile P sb)
proof –
from consis have sharing-consistent S O (takeWhile P sb @ dropWhile P sb)
by simp
with sharing-consistent-append [of - - takeWhile P sb dropWhile P sb]
show ?thesis
by simp
qed

lemma sharing-consis-tl: sharing-consis S (t#ts) ⇒ sharing-consis S ts
by (auto simp add: sharing-consis-def)

lemma sharing-consis-Cons:
[sharing-consis S ts; sharing-consistent S O sb]
⇒ sharing-consis S ((p,is,θ,sb,D,O,R)#ts)
apply (clarsimp simp add: sharing-consis-def)
subgoal for i pa isa O' R' D' θ' sba
by (case-tac i) auto
done
lemma outstanding-non-volatile-writes-unshared-tl:
outstanding-non-volatile-writes-unshared $S (t\#ts) \implies$
outstanding-non-volatile-writes-unshared $S ts$
by (auto simp add: outstanding-non-volatile-writes-unshared-def)

lemma no-outstanding-write-to-read-only-memory-tl:
no-outstanding-write-to-read-only-memory $S (t\#ts) \implies$
o-outstanding-write-to-read-only-memory $S ts$
by (auto simp add: no-outstanding-write-to-read-only-memory-def)

lemma valid-ownership-and-sharing-tl:
valid-ownership-and-sharing $S (t\#ts) \implies valid-ownership-and-sharing S ts$
apply (clarsimp simp add: valid-ownership-and-sharing-def)
apply (auto intro: valid-ownership-tl
      outstanding-non-volatile-writes-unshared-tl
      no-outstanding-write-to-read-only-memory-tl
      sharing-consis-tl)
done

lemma non-volatile-owned-or-read-only-outstanding-non-volatile-writes:
$\bigwedge O S \_{\text{pending-write}}. [\text{non-volatile-owned-or-read-only pending-write } S O sb]$
implies
outstanding-refs is-non-volatile-Write$_{ab}$ sb $\subseteq O \cup$ all-acquired sb
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write$_{ab}$ volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True
from Cons.hyps [of True ($S \oplus_W R \ominus_A L) (O \cup A - R$)] Cons.prems
show ?thesis
by (auto simp add: Write$_{ab}$ True)
next
case False with Cons show ?thesis
by (auto simp add: Write$_{ab}$)
qed
next
case Read$_{ab}$ with Cons show ?thesis
by auto
next
case Prog$_{ab}$ with Cons show ?thesis
by auto
next
case (Ghost$_{ab}$ A L R W)
from Cons.hyps [of pending-write \((S \oplus_W R \ominus_A L) (O \cup A \ominus R)\)] Cons.prems
show \(?\)thesis
by (auto simp add: Ghost sb)
qed
qed

lemma (in outstanding-non-volatile-refs-owned-or-read-only)
outstanding-non-volatile-writes-owned:
assumes i-bound: \(i < \) length ts
assumes ts-i: \(ts!i = (p,is,\theta,\sb,D,O,R)\)
shows outstanding-refs is-non-volatile-Write \(sb \subseteq O \cup \) all-acquired \(sb\)
using non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF 
outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts-i]]
by blast

lemma non-volatile-reads-acquired-or-read-only:
\(\bigwedge O S. \text{[non-volatile-owned-or-read-only True } S O sb; \text{sharing-consistent } S O sb]\)
\implies outstanding-refs is-non-volatile-Read \(sb \subseteq O \cup \) all-acquired \(sb \cup \) read-only \(S\)
proof (induct sb)
case Nil thus \(?\)case by simp
next
case (Cons x sb)
show \(?\)case
proof (cases x)
case (Write \(sb \) volatile a sop v A L R W)
show \(?\)thesis
proof (cases volatile)
case True

from Cons.prems obtain non-vol: non-volatile-owned-or-read-only True \((S \oplus_W R \ominus_A L) (O \cup A \ominus R)\) sb and
A-shared-onws: \(A \subseteq \text{dom } S \cup O\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and R-owns: \(R \subseteq O\) and
consis\': sharing-consistent \((S \oplus_W R \ominus_A L) (O \cup A \ominus R)\) sb
by (clarsimp simp add: Write sb True )

from Cons.hyps [OF non-vol consis]
have hyp: outstanding-refs is-non-volatile-Read \(sb \subseteq O \cup A \ominus R \cup \) all-acquired \(sb \cup \) read-only \((S \oplus_W R \ominus_A L)\).
with R-owns A-R L-A
show \(?\)thesis
apply (clarsimp simp add: Write sb True )
apply (drule (1) rev-subsetD)

148
apply (auto simp add: in-read-only-convs split: if-split-asm)
done

next
case False with Cons show ?thesis
by (auto simp add: Write sb)
qed
next
case Read\sb with Cons show ?thesis
by auto
next
case Prog\sb with Cons show ?thesis
by auto
next
case (Ghost\sb A L R W)
from Cons. prems obtain non-vol: non-volatile-owned-or-read-only True \((S ⊕ W R ⊓ A L)\) \((O∪A−R)\) sb and
A-shared-onws: \(A ⊆ \text{dom } S∪O \text{ and } L-A: L ⊆ A \text{ and } A-R: A ∩ R = \{\}\) and R-owns:
R ⊆ O and
consis': sharing-consistent \((S ⊕ W R ⊓ A L)\) \((O∪A−R)\) sb
by (clarsimp simp add: Ghost\sb )

from Cons.hyps [OF non-vol consis']
have hyp: outstanding-refs is-non-volatile-Read\sb sb
⊆ O ∪ A − R ∪ all-acquired sb ∪ read-only \((S ⊕ W R ⊓ A L)\).
with R-owns A-R L-A
show ?thesis
apply (clarsimp simp add: Ghost\sb )
apply (drule (1) rev-subsetD)
apply (auto simp add: in-read-only-convs split: if-split-asm)
done
qed
qed

lemma non-volatile-reads-acquired-or-read-only-reads:
\(\bigwedge S\) pending-write. [non-volatile-owned-or-read-only pending-write \(S \ O \ sb\)]
⇒
outstanding-refs is-non-volatile-Read\sb sb ⊆ O ∪ all-acquired sb ∪ read-only-reads O sb
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write\sb volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True

149
from Cons.prems obtain non-vol: non-volatile-owned-or-read-only True \((S \oplus_W R \ominus A L) (O \cup A - R) \) sb
by (clarsimp simp add: Write\_sb True )

from Cons.hyps [OF non-vol ]
have hyp: outstanding.refs is-non-volatile-Read\_sb sb
\subseteq O \cup A - R \cup all-acquired sb \cup read-only-reads (O \cup A - R) sb.
then
show \?thesis
by (auto simp add: Write\_sb)
next
case False with Cons show \?thesis
by (auto simp add: Write\_sb)
qed
next
case Read\_sb with Cons show \?thesis
by auto
next
case Prog\_sb with Cons show \?thesis
by auto
next
case (Ghost\_sb A L R W)
from Cons.prems obtain non-vol: non-volatile-owned-or-read-only pending-write \((S \oplus_W R \ominus A L) (O \cup A - R) \) sb
by (clarsimp simp add: Ghost\_sb )

from Cons.hyps [OF non-vol ]
have hyp: outstanding.refs is-non-volatile-Read\_sb sb
\subseteq O \cup A - R \cup all-acquired sb \cup read-only-reads (O \cup A - R) sb.
then
show \?thesis
by (auto simp add: Ghost\_sb )
qed
qed

lemma non-volatile-owned-or-read-only-outstanding.refs:
\(\bigwedge O S \) pending-write. [non-volatile-owned-or-read-only pending-write S O sb]
\implies outstanding.refs (Not \circ is-volatile) sb \subseteq O \cup all-acquired sb \cup read-only-reads O sb
proof (induct sb)
case Nil thus \?case by simp
next
case (Cons x sb)
show \?case
proof (cases x)
case (Write\_sb volatile a sop v A L R W)
show \?thesis
qed

150
proof (cases volatile)
case True
  from Cons.hyps [of True \( (S \oplus W R \ominus A L) (O \cup A - R) \)] Cons.prems
  show ?thesis
  by (auto simp add: Write sb True)
next
  case False with Cons show ?thesis
  by (auto simp add: Write sb)
qed

next
  case Read sb with Cons show ?thesis
  by auto
next
  case Prog sb with Cons show ?thesis
  by auto
next
  case (Ghost sb A L R W)
  from Cons.prems [of pending-write \( (S \oplus W R \ominus A L) (O \cup A - R) \)] Cons.prems
  show ?thesis
  by (auto simp add: Ghost sb)
qed

lemma no-unacquired-write-to-read-only:
\( \Lambda S O. \ [\text{no-write-to-read-only-memory } S sb; \text{sharing-consistent } S O sb; \)
a \in \text{read-only } S; a \notin (O \cup \text{all-acquired sb}] \)
  \implies a \notin \text{outstanding-refs is-Write sb}
proof (induct sb)
case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a' sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True

      from Cons.prems obtain no-wrt: no-write-to-read-only-memory \( (S \oplus W R \ominus A L) \) sb and
      A-shared-onws: A \subseteq \text{dom } S \cup O \text{ and } \text{L-A: } L \subseteq A \text{ and } \text{A-R: } A \cap R = {} \text{ and } \text{R-owns: } R \subseteq O \text{ and } 
      consis't: sharing-consistent \( (S \oplus W R \ominus A L) (O \cup A - R) \) sb and
      a-ro: a \in \text{read-only } S \text{ and } 
      a-A: a \notin A \text{ and } a\text{-all-acq: } a \notin \text{all-acquired sb and } a\text{-owns: } a \notin O \text{ and } 
      a'-notin: a' \notin \text{read-only } S
      by ( simp add: Write sb True )

151
from \(a\)'-notin a-ro have neq-a-a': \(a \neq a'\)
by blast

from a-A a-all-acq a-owns
have a-notin': \(a \notin O \cup A - R \cup \text{all-acquired sb}\)
by auto
from a-ro L-A a-A R-owns a-owns
have \(a \in \text{read-only } (S \oplus W R \ominus A L)\)
by (auto simp add: in-read-only-convs split: if-split-asm)

from Cons.hyps [OF no-wrt consis' this a-notin']
have a \(\notin \text{outstanding-refs is-Write}_{ab}\) sb.
with neq-a-a'
show ?thesis
by (clarsimp simp add: Write sb True)

next
  case False with Cons
  show ?thesis
by (auto simp add: Write sb False)
qed

next
  case Read_{ab} with Cons
  show ?thesis
  by (auto)

next
  case Prog_{ab} with Cons
  show ?thesis
  by (auto)

next
  case (Ghost_{ab} A L R W)
  from Cons.prems obtain no-wrt: no-write-to-read-only-memory (\(S \oplus W R \ominus A L\)) sb and
  
  A-shared-onws: \(A \subseteq \text{dom } S \cup O \text{ and } L-A: L \subseteq A \text{ and } A-R: A \cap R = {} \text{ and } R\)-owns:
  
  \(R \subseteq O \text{ and } \text{consis'}\): sharing-consistent (\(S \oplus W R \ominus A L\)) (\(O \cup A - R\)) sb and
  
  a-ro: \(a \in \text{read-only } S \text{ and } \text{a-A: } a \notin A \text{ and } a\)-all-acq: \(a \notin \text{all-acquired sb } \text{and } a\)-owns: \(a \notin O\)
  
  by (simp add: Ghost_{sb})

from a-A a-all-acq a-owns
have a-notin': \(a \notin O \cup A - R \cup \text{all-acquired sb}\)
by auto
from a-ro L-A a-A R-owns a-owns
have \(a \in \text{read-only } (S \oplus W R \ominus A L)\)
by (auto simp add: in-read-only-convs split: if-split-asm)

from Cons.hyps [OF no-wrt consis' this a-notin']
have a \(\notin \text{outstanding-refs is-Write}_{ab}\) sb.
then
show ?thesis
by (clarsimp simp add: Ghost sb)
qed

lemma read-only-reads-read-only:
\[ \bigwedge S O. \text{[non-volatile-owned-or-read-only True } S O \text{ sb; }
\text{sharing-consistent } S O \text{ sb]} \]
\[ \implies \text{read-only-reads } O \text{ sb } \subseteq O \cup \text{ all-acquired sb } \cup \text{ read-only } S \]

proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True

from Cons.prems obtain non-vol: non-volatile-owned-or-read-only True \((S \oplus_W R \ominus_A L)\) \((O \cup A - R)\) sb and
A-shared-ons: A \(\subseteq\) dom S \(\cup\) O and L-A: L \(\subseteq\) A and A-R: A \(\cap\) R = {} and R-owns:
R \(\subseteq\) O and
consis': sharing-consistent \((S \oplus_W R \ominus_A L)\) \((O \cup A - R)\) sb
by (clarsimp simp add: Write sb True)

from Cons.hyps [OF non-vol consis']
have hyp: read-only-reads \((O \cup A - R)\) sb
\(\subseteq\) O \(\cup\) A - R \(\cup\) all-acquired sb \(\cup\) read-only \((S \oplus_W R \ominus_A L)\).

{ fix a' 
  assume a'-in: a' \(\in\) read-only-reads \((O \cup A - R)\) sb
  assume a'-unowned: a' \(\notin\) O
  assume a'-unacq: a' \(\notin\) all-acquired sb
  assume a'-A: a' \(\notin\) A
  have a' \(\in\) read-only S
  proof
    from a'-in hyp a'-unowned a'-unacq a'-A
    have a' \(\in\) read-only \((S \oplus_W R \ominus_A L)\)
    by auto
  
  with L-A R-owns a'-unowned
  show ?thesis
    by (auto simp add: in-read-only-convs split:if-split-asm)
  qed
}
then

show ?thesis
apply (clarsimp simp add: Write sb True simp del: o-apply)
apply force
done
next
case False with Cons show ?thesis
by (auto simp add: Write sb)
qed
next
case Read sb with Cons show ?thesis
by auto
next
case Prog sb with Cons show ?thesis
by auto
next
case (Ghost sb A L R W)
from Cons.prems obtain non-vol: non-volatile-owned-or-read-only True (S ⊕ W R ⊕ A L) (O ∪ A − R) sb and
A-shared-onws: A ⊆ dom S ∪ O and L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns:
R ⊆ O and
consis': sharing-consistent (S ⊕ W R ⊕ A L) (O ∪ A − R) sb
by (clarsimp simp add: Ghost sb)

from Cons.hyps [OF non-vol consis']
have hyp: read-only-reads (O ∪ A − R) sb
⊆ O ∪ A − R ∪ all-acquired sb ∪ read-only (S ⊕ W R ⊕ A L).

{|}
fix a'
assume a'-in: a' ∈ read-only-reads (O ∪ A − R) sb
assume a'-unowned: a' ∉ O
assume a'-unacq: a' ∉ all-acquired sb
assume a'-A: a' ∉ A
have a' ∈ read-only S
proof −
from a'-in hyp a'-unowned a'-unacq a'-A
have a' ∈ read-only (S ⊕ W R ⊕ A L)
by auto

with L-A R-owns a'-unowned
show ?thesis
by (auto simp add: in-read-only-convs split:if-split-asym)
qed
|
then

show ?thesis

154
apply (clarsimp simp add: Ghost sb simp del: o-apply)
apply force
done

qed
qed

lemma no-unacquired-write-to-read-only-reads:
\[
\forall S \ O . \ [\text{no-write-to-read-only-memory } S \ sb; \non-volatile-owned-or-read-only \ True \ S \ O \ sb; \sharing-consistent \ S \ O \ sb; \ a \in \text{read-only-reads } O \ sb; \ a \notin (O \cup \text{all-acquired } sb)] 
\implies a \notin \text{outstanding-refs } \text{is-Write}_{sb}
\]

proof (induct sb)
case Nil thus \?case by simp

next

case (Cons x sb)
show \?case
proof (cases x)
case (Write sb volatile a' sop v A L R W)
show \?thesis
proof (cases volatile)
case True
from Cons.prems obtain no-wrt: no-write-to-read-only-memory \((S \oplus W R \ominus A L) sb\) and
non-vol: non-volatile-owned-or-read-only True \((S \oplus W R \ominus A L) (O \cup A - R) sb\) and
A-shared-ons: \(A \subseteq \text{dom } S \cup O \text{ and } L-A: L \subseteq A \text{ and } A-R: A \cap R = \{\} \text{ and } R-owns: R \subseteq O \text{ and }
\) sharing-consistent \((S \oplus W R \ominus A L) (O \cup A - R) sb\) and
a-ro: \(a \in \text{read-only-reads } (O \cup A - R) sb\) and
a-A: \(a \notin A \text{ and } a\text{-all-acq: } a \notin \text{all-acquired } sb \text{ and } a\text{-owns: } a \notin O \text{ and }
a'\text{-notin: } a' \notin \text{read-only } S\)
by ( simp add: Write sb True )

from read-only-reads-read-only \[OF \text{non-vol } \text{consis'}\] a-ro a-owns a-all-acq a-A
have \(a \in \text{read-only } (S \oplus W R \ominus A L)\)
by auto
with a'-notin R-owns a-owns have neq-a-a': a\#a'
by (auto simp add: in-read-only-convs split: if-split-asm)

from a-A a-all-acq a-owns
have a-notin': \(a \notin O \cup A - R \cup \text{all-acquired sb}\)
by auto

from Cons.hyps \[OF \text{no-wrt } \text{non-vol } \text{consis'} \text{ a-ro } a\text{-notin'}\]
have \(a \notin \text{outstanding-refs } \text{is-Write}_{sb}\)
then
show \?thesis
using neq-a-a'
by (auto simp add: Write sb True)
next
case False with Cons
show \?thesis
by (auto simp add: Write sb False)
qed
next
case (Read sb volatile a' t v)
show \?thesis
proof (cases volatile)
case True
with Cons show \?thesis
by (clarsimp simp add: Read sb)
next
case False
note non-volatile = this
from Cons.prems obtain no-wrt': no-write-to-read-only-memory \( S \) \( \sigma \) \( Sb \) and
consis'sharing-consistent \( S \) \( \sigma \) \( Sb \) and
a-in: a ∈ (if a' ∉ \( \sigma \) then insert a' (read-only-reads \( \sigma \) \( Sb \))
else read-only-reads \( \sigma \) \( Sb \)) and
a'-owns-shared: a' ∈ \( \sigma \) ∨ a' ∈ read-only \( S \) and
non-vol': non-volatile-owned-or-read-only True \( S \) \( \sigma \) \( Sb \) and
a-owns: a ∉ \( \sigma \) ∪ all-acquired \( Sb \)
by (clarsimp simp add: Read sb False)

show \?thesis
proof (cases a' ∈ \( \sigma \))
case True
with a-in have a ∈ read-only-reads \( \sigma \) \( Sb \)
by auto
from Cons.hyps [OF no-wrt' non-vol' consis' this a-owns]
show \?thesis
by (clarsimp simp add: Read sb)
next
case False
note a'-unowned = this
with a-in have a-in': a ∈ insert a' (read-only-reads \( \sigma \) \( Sb \)) by auto
from a'-owns-shared False have a'-read-only: a' ∈ read-only \( S \) by auto
show \?thesis
proof (cases a=a')
case False
with a-in' have a ∈ (read-only-reads \( \sigma \) \( Sb \)) by auto
from Cons.hyps [OF no-wrt' non-vol' consis' this a-owns]
show \?thesis
by (clarsimp simp add: Read sb)
next
case True
from no-unacquired-write-to-read-only [OF no-wrt' consis' a'-read-only] a-owns True
have a' ∉ outstanding-refs is-Write sb
by auto

then show ?thesis
  by (simp add: "Read" True)

qed

qed

qed

next
  case Prog with Cons
  show ?thesis
    by (auto)

next
  case (Ghost A L R W)
  from Cons.prems obtain no-wrt: no-write-to-read-only-memory (S ⊕ W R ⊏ A L) sb and
    non-vol: non-volatile-owned-or-read-only True (S ⊕ W R ⊏ A L) (O ⊎ A − R) sb and
    A-shared-onws: A ⊆ dom S ⊎ O and L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns:
      R ⊆ O and
      consis': sharing-consistent (S ⊕ W R ⊏ A L) (O ⊎ A − R) sb and
      a-ro: a ∈ read-only-reads (O ⊎ A − R) sb and
      a-A: a ∉ A and a-all-acq: a ∉ all-acquired sb and
      a-owns: a ∉ O
    by ( simp add: Ghost)
  from read-only-reads-read-only [OF non-vol consis'] a-ro a-owns a-all-acq a-A
  have a ∈ read-only (S ⊕ W R ⊏ A L)
    by auto

  from a-A a-all-acq a-owns
  have a-notin': a ∉ O ⊎ A − R ∪ all-acquired sb
    by auto

  from Cons.hyps [OF non-wrt non-vol consis' a-ro a-notin']
  have a ∉ outstanding-refs is-Write sb.
  then
  show ?thesis
    by (auto simp add: Ghost)

qed

qed

lemma no-unacquired-write-to-read-only′′:
  assumes no-wrt: no-write-to-read-only-memory (S ⊕ W R ⊏ A L) sb
  assumes consis: sharing-consistent S O sb
  shows read-only (S ∩ outstanding-refs is-Write sb) ⊆ O ∪ all-acquired sb
using no-unacquired-write-to-read-only [OF no-wrt consis]
by auto

lemma no-unacquired-volatile-write-to-read-only:
  assumes no-wrt: no-write-to-read-only-memory (S ⊕ W R ⊏ A L) sb
  assumes consis: sharing-consistent S O sb
shows read-only $S \cap$ outstanding-refs is-volatile-Write$_{sb}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb

proof –
  have outstanding-refs is-volatile-Write$_{sb}$ sb $\subseteq$ outstanding-refs is-Write$_{sb}$ sb
  apply (rule outstanding-refs-mono-pred)
  apply (auto simp add: is-volatile-Write$_{sb}$-def split: memref.splits)
  done
  with no-unacquired-write-to-read-only" [OF no-wrt consis]
  show ?thesis by blast
qed

lemma no-unacquired-non-volatile-write-to-read-only-reads:
  assumes no-wrt: no-write-to-read-only-memory $S$ sb
  assumes consis: sharing-consistent $S \mathcal{O}$ sb
  shows read-only $S \cap$ outstanding-refs is-non-volatile-Write$_{sb}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb
  proof –
  from outstanding-refs-subsets
  have outstanding-refs is-non-volatile-Write$_{sb}$ sb $\subseteq$ outstanding-refs is-Write$_{sb}$ sb by –
  assumption
  with no-unacquired-write-to-read-only" [OF no-wrt consis]
  show ?thesis by blast
qed

lemma no-unacquired-write-to-read-only-reads₃:
  assumes no-wrt: no-write-to-read-only-memory $S$ sb
  assumes non-vol: non-volatile-owned-or-read-only True $S \mathcal{O}$ sb
  assumes consis: sharing-consistent $S \mathcal{O}$ sb
  shows read-only-reads $\mathcal{O}$ sb $\cap$ outstanding-refs is-Write$_{sb}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb
  using no-unacquired-write-to-read-only-reads [OF no-wrt non-vol consis]
  by auto

lemma no-unacquired-volatile-write-to-read-only-reads:
  assumes no-wrt: no-write-to-read-only-memory $S$ sb
  assumes non-vol: non-volatile-owned-or-read-only True $S \mathcal{O}$ sb
  assumes consis: sharing-consistent $S \mathcal{O}$ sb
  shows read-only-reads $\mathcal{O}$ sb $\cap$ outstanding-refs is-volatile-Write$_{sb}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb
  proof –
  have outstanding-refs is-volatile-Write$_{sb}$ sb $\subseteq$ outstanding-refs is-Write$_{sb}$ sb
  apply (rule outstanding-refs-mono-pred)
  apply (auto simp add: is-volatile-Write$_{sb}$-def split: memref.splits)
  done
  with no-unacquired-write-to-read-only-reads [OF no-wrt non-vol consis]
  show ?thesis by blast
qed

lemma no-unacquired-non-volatile-write-to-read-only:
  assumes no-wrt: no-write-to-read-only-memory $S$ sb
  assumes non-vol: non-volatile-owned-or-read-only True $S \mathcal{O}$ sb
  assumes consis: sharing-consistent $S \mathcal{O}$ sb

158
shows read-only-reads $\mathcal{O}$ sb $\cap$ outstanding-refs is-non-volatile-$\text{Write}_{sb}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb

proof –

from outstanding-refs-subsets

have outstanding-refs is-non-volatile-$\text{Write}_{sb}$ sb $\subseteq$ outstanding-refs is-$\text{Write}_{sb}$ sb by – assumption

with no-unacquired-write-to-read-only-reads [OF no-wrt non-vol consis]

show thesis by blast

qed

lemma set-dropWhileD: $x \in \text{set} (\text{dropWhile} \ P \ xs) \implies x \in \text{set} \ xs$

by (induct xs) (auto split: if-split-asm)

lemma outstanding-refs-takeWhileD:

$x \in \text{outstanding-refs} \ P \ (\text{takeWhile} \ P' \ sb) \implies x \in \text{outstanding-refs} \ P \ sb$

using outstanding-refs-takeWhile

by blast

lemma outstanding-refs-dropWhileD:

$x \in \text{outstanding-refs} \ P \ (\text{dropWhile} \ P' \ sb) \implies x \in \text{outstanding-refs} \ P \ sb$

by (auto dest: set-dropWhileD simp add: outstanding-refs-conv)

lemma dropWhile-ConsD: dropWhile $P$ $xs = y \# ys \implies \neg P \ y$

by (simp add: dropWhile-eq-Cons-conv)

lemma non-volatile-owned-or-read-only-drop:

non-volatile-owned-or-read-only $False \ S \ O$ sb

implies non-volatile-owned-or-read-only $True$

(share (takeWhile $\neg \circ \text{is-volatile-Write}_{sb}$ sb) $S$)

(acquired True (takeWhile $\neg \circ \text{is-volatile-Write}_{sb}$ sb) $O$)

(dropWhile (Not $\circ \text{is-volatile-Write}_{sb}$ sb) sb)

using non-volatile-owned-or-read-only-append [of $False \ S \ O$ (takeWhile $\neg \circ \text{is-volatile-Write}_{sb}$ sb)]

apply (cases outstanding-refs is-volatile-$\text{Write}_{sb}$ sb $= \{\}$)

apply (clarsimp simp add: outstanding-vol-write-take-drop-appends

takeWhile-not-vol-write-outstanding-refs dropWhile-not-vol-write-empty)

apply (clarsimp simp add: outstanding-vol-write-take-drop-appends

takeWhile-not-vol-write-outstanding-refs dropWhile-not-vol-write-empty)

apply (case-tac (dropWhile $\neg \circ \text{is-volatile-Write}_{sb}$ sb))

apply (fastforce simp add: outstanding-refs-conv)

apply (frule dropWhile-ConsD)

apply (clarsimp split: memref.splits)

done

159
lemma  read-only-share: \(\bigwedge S \ O\).  
sharing-consistent \(S \ O\) \(sb \implies \)
  read-only (share sb \(S\)) \(\subseteq\) read-only \(S \cup O \cup\) all-acquired \(sb\)
proof (induct \(sb\))
case Nil thus ?case by auto
next
case (Cons \(x\) \(sb\))
show ?case
proof (cases \(x\))
case (Write\(_{ab}\) volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True
from Cons.prems obtain
A-shared-owns: \(A \subseteq \text{dom} S \cup O\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and R-owns:
\(R \subseteq O\) and
consis': sharing-consistent \((S \oplus_{W} R \ominus_{A} L)\) \((O \cup A - R)\) \(sb\)
by (clarsimp simp add: Write\(_{ab}\) True )
from Cons.hyps [OF consis']
have read-only (share \(sb\) \((S \oplus_{W} R \ominus_{A} L)\))
  \(\subseteq\) read-only \((S \oplus_{W} R \ominus_{A} L)\) \((O \cup A - R)\) \(\cup\) all-acquired \(sb\)
by auto
also from A-shared-owns L-A R-owns A-R
have read-only \((S \oplus_{W} R \ominus_{A} L)\) \((O \cup A - R)\) \(\cup\) all-acquired \(sb\) \(\subseteq\)
read-only \(S \cup O \cup (A \cup\) all-acquired \(sb)\)
by (auto simp add: read-only-def augment-shared-def restrict-shared-def split: option.splits)
finally
thesis
by (simp add: Write\(_{ab}\) True)
next
case False with Cons show ?thesis
by (auto simp add: Write\(_{ab}\))
qed
next
case Read\(_{ab}\) with Cons show ?thesis
by auto
next
case Prog\(_{ab}\) with Cons show ?thesis
by auto
next
case (Ghost\(_{ab}\) A L R W)
from Cons.prems obtain
A-shared-owns: \(A \subseteq \text{dom} S \cup O\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and R-owns:
\(R \subseteq O\) and
consis': sharing-consistent \((S \oplus_{W} R \ominus_{A} L)\) \((O \cup A - R)\) \(sb\)
by (clarsimp simp add: Ghost\(_{ab}\) )
from Cons.hyps [OF consis']
have read-only (share \(sb\) \((S \oplus_{W} R \ominus_{A} L)\))
\[ \subseteq \text{read-only } (S \oplus W R \ominus A L) \cup (O \cup A - R) \cup \text{all-acquired sb} \]

by auto

also from A-shared-owns L-A R-owns A-R

have read-only \((S \oplus W R \ominus A L) \cup (O \cup A - R) \cup \text{all-acquired sb} \subseteq \)

read-only \(S \cup O \cup (A \cup \text{all-acquired sb}) \)

by (auto simp add: read-only-def augment-shared-def restrict-shared-def split: option.splits)

finally

show \(?\text{thesis}\)

by (simp add: Ghost sb)

qed

qed

lemma \textbf{(in } \text{valid-ownership-and-sharing) outstanding-non-write-non-vol-reads-drop-disj:}

assumes i-bound: \(i < \text{length ts}\)

assumes j-bound: \(j < \text{length ts}\)

assumes neq-i-j: \(i \neq j\)

assumes ith: ts!i = \((p_i, is_i, \theta_i, sb_i, D_i, O_i, R_i)\)

assumes jth: ts!j = \((p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j)\)

shows outstanding-refs is-Write sb (dropWhile (Not ◦ is-volatile-Write sb) sb_i) ∩

outstanding-refs is-non-volatile-Read sb (dropWhile (Not ◦ is-volatile-Write sb) sb_j)

= \(\{\}\)

proof –

let \(\text{?take-j} = (\text{takeWhile (Not ◦ is-volatile-Write sb) sb_j})\)

let \(\text{?drop-j} = (\text{dropWhile (Not ◦ is-volatile-Write sb) sb_j})\)

let \(\text{?take-i} = (\text{takeWhile (Not ◦ is-volatile-Write sb) sb_i})\)

let \(\text{?drop-i} = (\text{dropWhile (Not ◦ is-volatile-Write sb) sb_i})\)

note nvo-i = outstanding-non-volatile-refs-owned-or-read-only \([\text{OF } i\text{-bound } \text{ith}]\)

note nvo-j = outstanding-non-volatile-refs-owned-or-read-only \([\text{OF } j\text{-bound } \text{jth}]\)

note nvo-i = outstanding-non-volatile-ref-owned-or-read-only \([\text{OF } i\text{-bound } \text{ith}]\)

with no-write-to-read-only-memory-append \([\text{OF } S \text{ ?take-i } \text{?drop-i}]\)

have nro-drop-i: no-write-to-read-only-memory (share ?take-i S) ?drop-i

by simp

note nro-j = no-outstanding-write-to-read-only-memory \([\text{OF } j\text{-bound } \text{jth}]\)

with no-write-to-read-only-memory-append \([\text{OF } S \text{ ?take-j } \text{?drop-j}]\)

have nro-drop-j: no-write-to-read-only-memory (share ?take-j S) ?drop-j

by simp

from outstanding-volatile-writes-unowned-by-others \([\text{OF } i\text{-bound } j\text{-bound } \text{neq-i-j } \text{ith } \text{jth}]\)

have dist: \((O_j \cup \text{all-acquired sb}_j) \cap \text{outstanding-refs is-volatile-Write sb}_i = \{\}\)

note own-dist = ownership-distinct \([\text{OF } i\text{-bound } j\text{-bound } \text{neq-i-j } \text{ith } \text{jth}]\)
from sharing-consis [OF j-bound jth]
have consis-j: sharing-consistent $\mathcal{S} \ O_j \ sb_j$.
with sharing-consistent-append [of $\mathcal{S} \ O_j \ ?take-j \ ?drop-j$
obtain
  consis-take-j: sharing-consistent $\mathcal{S} \ O_j \ ?take-j$ and
  consis-drop-j: sharing-consistent (share $?take-j \ S$) (acquired True $?take-j \ O_j$) $?drop-j$
  by simp

from sharing-consis [OF i-bound ith]
have consis-i: sharing-consistent $\mathcal{S} \ O_i \ sb_i$.
with sharing-consistent-append [of $\mathcal{S} \ O_i \ ?take-i \ ?drop-i$
have consis-drop-i: sharing-consistent (share $?take-i \ S$) (acquired True $?take-i \ O_i$) $?drop-i$
  by simp

{
  fix x
  assume x-in-drop-i: $x \in$ outstanding-refs is-Write$_{sb}$ $?drop-i$
  assume x-in-drop-j: $x \in$ outstanding-refs is-non-volatile-Read$_{sb}$ $?drop-j$
  have False
  proof
    from x-in-drop-i have x-in-i: $x \in$ outstanding-refs is-Write$_{sb}$ $sb_i$
    using outstanding-refs-append [of is-Write$_{sb}$ $?take-i \ ?drop-i$] by auto

    from x-in-drop-j have x-in-j: $x \in$ outstanding-refs is-non-volatile-Read$_{sb}$ $sb_j$
    using outstanding-refs-append [of is-non-volatile-Read$_{sb}$ $?take-j \ ?drop-j$]
    by auto
    from non-volatile-owned-or-read-only-drop [OF nvo-j]
    have nvo-drop-j: non-volatile-owned-or-read-only True (share $?take-j \ S$) (acquired True $?take-j \ O_j$) $?drop-j$
    using all-acquired-append [of $?take-j \ ?drop-j$]
    by ( auto )

    from non-volatile-reads-acquired-or-read-only-reads [OF nvo-drop-j $x$-in-drop-j
    acquired-takeWhile-non-volatile-Write$_{sb}$ [of $sb_j \ O_j$]
    have x-j: $x \in O_j \cup$ all-acquired $sb_j \cup$ read-only-reads (acquired True $?take-j \ O_j$) $?drop-j$
    using all-acquired-append [of $?take-j \ ?drop-j$]
    by ( auto )

  assume x-in-vol-drop-i: $x \in$ outstanding-refs is-volatile-Write$_{sb}$ $?drop-i$
  hence x-in-vol-i: $x \in$ outstanding-refs is-volatile-Write$_{sb}$ $sb_i$
  using outstanding-refs-append [of is-volatile-Write$_{sb}$ $?take-i \ ?drop-i$]
  by auto

  from outstanding-volatile-writes-unowned-by-others [OF i-bound j-bound neq-i-j ith jth]
  have ($O_j \cup$ all-acquired $sb_j$) $\cap$ outstanding-refs is-volatile-Write$_{sb}$ $sb_i$ = $\{\}$.
  with x-in-vol-i $x$-j obtain
    x-unacq-j: $x \notin O_j \cup$ all-acquired $sb_j$ and
    x-ror-j: $x \in$ read-only-reads (acquired True $?take-j \ O_j$) $?drop-j$
    by auto

162
from read-only-reads-unowned \([\text{OF } j\text{-bound } i\text{-bound } \text{neq-i-j } \text{symmetric } j\text{th } i\text{th}]\) \(x\text{-ror-j}\)
have \(x \not\in \mathcal{O}_i \cup \text{all-acquired } sb_i\)
by auto

moreover

from read-only-reads-read-only \([\text{OF } nvo\text{-drop-j } \text{consis-drop-j}]\) \(x\text{-ror-j} \ x\text{-unacq-j}\)
all-acquired-append \([\text{of } ?\text{take-j } ?\text{drop-j}]\) acquired-takeWhile-non-volatile-Write_{sb} \([\text{of } sb_j \ \mathcal{O}_j]\)
have \(x \in \text{read-only } (\text{share } ?\text{take-j } S)\)
by (auto)

from read-only-share \([\text{OF } \text{consis-take-j}]\) this \(x\text{-unacq-j} \ \text{all-acquired-append } [\text{of } ?\text{take-j } ?\text{drop-j}]\)

have \(x \in \text{read-only } S\)
by auto

with no-unacquired-write-to-read-only” \([\text{OF } nro\text{-i } \text{consis-i}]\) \(x\text{-in-i}\)
have \(x \in \mathcal{O}_i \cup \text{all-acquired } sb_i\)
by auto

ultimately have False by auto
}

moreover
{
assume x-in-non-vol-drop-i: \(x \in \text{outstanding-refs is-non-volatile-Write}_{sb} \ ?\text{drop-i}\)
hence \(x \in \text{outstanding-refs is-non-volatile-Write}_{sb} \ sb_i\)
using outstanding-refs-append \([\text{of is-non-volatile-Write}_{sb} \ ?\text{take-i } ?\text{drop-i}]\)
by auto
with non-volatile-owned-or-read-only-outstanding-non-volatile-writes \([\text{OF } nvo\text{-i}]\)
have \(x \in \mathcal{O}_i \cup \text{all-acquired } sb_i\) by auto

moreover

with \(x\text{-j own-dist obtain}\)
\(x\text{-unacq-j}: \(x \not\in \mathcal{O}_j \cup \text{all-acquired } sb_j\) \text{ and}\)
\(x\text{-ror-j}: \(x \in \text{read-only-reads } (\text{acquired } \text{True } ?\text{take-j } \mathcal{O}_j) \ ?\text{drop-j}\)
by auto
from read-only-reads-unowned \([\text{OF } j\text{-bound } i\text{-bound } \text{neq-i-j } \text{symmetric } j\text{th } i\text{th}]\) \(x\text{-ror-j}\)
have \(x \not\in \mathcal{O}_i \cup \text{all-acquired } sb_i\)
by auto

ultimately have False
by auto
}
ultimately

show ?thesis
using x-in-drop-i x-in-drop-j

163
by (auto simp add: misc-outstanding-refs-convs)
   qed

{ 
thus ?thesis
   by auto

qed

lemma (in valid-ownership-and-sharing) outstanding-non-volatile-write-disj:
  assumes i-bound: i < length ts
  assumes j-bound: j < length ts
  assumes neq-i-j: i ≠ j
  assumes ith: ts!i = (p_i, i_s, θ_i, sb_i, D_i, O_i, R_i)
  assumes jth: ts!j = (p_j, i_s, θ_j, sb_j, D_j, O_j, R_j)
  shows outstanding-refs (is-non-volatile-Write sb_i ∩
    (outstanding-refs is-volatile-Write sb_i ∪
     outstanding-refs is-non-volatile-Write sb_i ∪
     outstanding-refs is-non-volatile-Read sb_i)
    (takeWhile (Not ◦ is-volatile-Write sb_i) sb_i)
    ∪
    (outstanding-refs is-non-volatile-Read sb_i (takeWhile (Not ◦ is-volatile-Write sb_i) sb_i)
    −
    read-only-reads O_j (takeWhile (Not ◦ is-volatile-Write sb_j) sb_j)) ∪
    (O_j ∪ all-acquired (takeWhile (Not ◦ is-volatile-Write sb_j) sb_j))
  ) = { } (is ?non-vol-writes-i ∩ ?not-volatile-j = { })

proof –
  note nro-i = no-outstanding-write-to-read-only-memory [OF i-bound ith]
  note nro-j = no-outstanding-write-to-read-only-memory [OF j-bound jth]
  note nvo-j = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound jth]
  note nvo-i = outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ith]

  from outstanding-volatile-writes-unowned-by-others [OF i-bound j-bound neq-i-j ith jth] 
  have dist: (O_j ∪ all-acquired sb_j) ∩ outstanding-refs is-volatile-Write sb_i = {}.

  from outstanding-volatile-writes-unowned-by-others [OF j-bound i-bound neq-i-j
  [symmetric] jth ith] 
  have dist-j: (O_i ∪ all-acquired sb_i) ∩ outstanding-refs is-volatile-Write sb_j = {}.

  note own-dist = ownership-distinct [OF i-bound j-bound neq-i-j ith jth]

  from sharing-consis [OF j-bound jth]
  have consis-j: sharing-consistent $\mathcal{S}$ O_j sb_j.

  from sharing-consis [OF i-bound ith]
  have consis-i: sharing-consistent $\mathcal{S}$ O_i sb_i.

  let ?take-j = (takeWhile (Not ◦ is-volatile-Write sb_i) sb_j)
  let ?drop-j = (dropWhile (Not ◦ is-volatile-Write sb_i) sb_j)

  { 

  164
fix x
assume x-in-take-i: x ∈ ?non-vol-writes-i
assume x-in-j: x ∈ ?not-volatile-j
from x-in-take-i have x-in-i: x ∈ outstanding-refs (is-non-volatile-Write sb_i) by (auto dest: outstanding-refs-takeWhileD)
from non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF nvo-i] x-in-i have x-in-owns-acq-i: x ∈ O_i ∪ all-acquired sb_i by auto
have False
proof –

{ assume x-in-j: x ∈ outstanding-refs is-volatile-Write sb_j with dist-j have x-notin: x /∈ (O_i ∪ all-acquired sb_i) by auto with x-in-owns-acq-i have False by auto }
moreover

{ assume x-in-j: x ∈ outstanding-refs is-non-volatile-Write sb_j from non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF nvo-j] x-in-j have x ∈ O_j ∪ all-acquired sb_j by auto with x-in-owns-acq-i own-dist have False by auto }
moreover

{ assume x-in-j: x ∈ outstanding-refs is-non-volatile-Read sb_j
from non-volatile-owned-or-read-only-drop [OF nvo-j] have nvo': non-volatile-owned-or-read-only True (share ?take-j S) (acquired True ?take-j O_j) ?drop-j.
from non-volatile-owned-or-read-only-outstanding-refs [OF nvo'] x-in-j have x ∈ acquired True ?take-j O_j ∪ all-acquired ?drop-j ∪ read-only-reads (acquired True ?take-j O_j) ?drop-j by (auto simp add: misc-outstanding-refs-conv)

ultimately
with read-only-reads-unowned [OF j-bound i-bound neq-i-j [symmetric] jth ith]
x-in-owns-acq-i
have False
  by auto
}
moreover
{
assume x-in-j: x ∈ outstanding-refs is-non-volatile-Read_{sb} ?take-j
assume x-notin: x \notin read-only-reads \var{O}_j \ ?take-j
from non-volatile-owned-or-read-only-append [where xs=?take-j and ys=?drop-j] nvo-j
have non-volatile-owned-or-read-only False S \var{O}_j \ ?take-j
  by auto
from non-volatile-owned-or-read-only-outstanding-refs [OF this] x-in-j x-notin
have x ∈ \var{O}_j \cup all-acquired ?take-j
  by (auto simp add: misc-outstanding-refs-convs)
have False
  by auto
}
moreover
{
assume x-in-j: x ∈ \var{O}_j \cup all-acquired ?take-j
moreover
from all-acquired-append [of ?take-j ?drop-j]
have all-acquired ?take-j ⊆ all-acquired sb_{j}
  by auto
ultimately have False
  using x-in-owns-acq-i own-dist
  by auto
}
ultimately show ?thesis
using x-in-take-i x-in-j
by (auto simp add: misc-outstanding-refs-convs)
qed
}
then show ?thesis
  by auto
qed

lemma (in valid-ownership-and-sharing) outstanding-non-volatile-write-not-volatile-read-disj:
assumes i-bound: i < length ts
assumes j-bound: j < length ts
assumes neq-i-j: i \neq j
assumes ith: ts!i = (p_i, is_i, \delta_i, sb_i, D_i, O_i, R_i)
assumes jth: ts!j = (p_j, is_j, \delta_j, sb_j, D_j, O_j, R_j)
shows outstanding-refs (is-non-volatile-Write_{sb}) (takeWhile (Not o is-volatile-Write_{sb}) sb_{j}) ∩
outstanding-refs (Not ◦ is-volatile-Read sb) (dropWhile (Not ◦ is-volatile-Write sb) sb) = {}
(is ?non-vol-writes-i ∩ ?not-volatile-j = {})

proof
have outstanding-refs (Not ◦ is-volatile-Read sb) (dropWhile (Not ◦ is-volatile-Write sb) sb) ⊆
  outstanding-refs is-volatile-Write sb ∪
  outstanding-refs is-non-volatile-Write sb ∪
  outstanding-refs is-non-volatile-Read sb (dropWhile (Not ◦ is-volatile-Write sb) sb)
by (auto simp add: misc-outstanding-refs-convs dest: outstanding-refs-dropWhileD)
with outstanding-non-volatile-write-disj [OF i-bound j-bound neq-i-j ith jth]
show ?thesis
  by blast
qed

lemma (in valid-ownership-and-sharing) outstanding-refs-is-Write sb-takeWhile-disj:
  \forall i < length ts. (\forall j < length ts. i \neq j \rightarrow
    \begin{align*}
      \text{(let \((\cdot,\cdot,\cdot,\cdot,sb_i,\cdot,\cdot,\cdot) = ts!i;)}
      \text{(\cdot,\cdot,\cdot,sb_j,\cdot,\cdot,\cdot) = ts!j)}
    \end{align*}
  in outstanding-refs is-Write sb \cap
  outstanding-refs is-Write (takeWhile (Not ◦ is-volatile-Write sb) sb) =

proof
{\begin{align*}
  \text{fix } i \ j \ p_i \ is_i \ O_i \ R_i \ D_i \ \check{v}_i \ sb_i \ p_j \ is_j \ O_j \ R_j \ D_j \ \check{v}_j \ sb_j
  \text{assume i-bound: } i < \text{length ts}
  \text{assume j-bound: } j < \text{length ts}
  \text{assume neq-i-j: } i \neq j
  \text{assume ith: } ts!i = (p_i,is_i,\check{v}_i,\cdot,sb_i,\cdot,D_i,O_i,R_i)
  \text{assume jth: } ts!j = (p_j,is_j,\check{v}_j,\cdot,sb_j,\cdot,D_j,O_j,R_j)
  \text{from outstanding-non-volatile-write-disj [OF j-bound i-bound neq-i-j[symmetric] jth ith]}
  \text{have outstanding-refs is-Write sb \cap}
    \text{outstanding-refs is-Write (takeWhile (Not ◦ is-volatile-Write sb) sb) = }\{\}
  \text{apply (clarsimp simp add: outstanding-refs-is-non-volatile-Write-takeWhile-conv)}
  \text{done}
\end{align*}}
thus ?thesis
  by (fastforce simp add: Let-def)
qed

fun read-tmps:: 'p store-buffer ⇒ tmp set
where
  read-tmps [] = {}
| read-tmps (r#rs) =

167
(case r of
  Read\textsubscript{ab} volatile a t v ⇒ insert t (read-tmps rs)
| - ⇒ read-tmps rs)

**Lemma** in-read-tmps-conv:
\((t ∈ \text{read-tmps } xs) = (\exists \text{volatile } a v. \text{Read}\textsubscript{ab} \text{volatile } a t v ∈ \text{set } xs)\)
by (induct xs) (auto split: memref.splits)

**Lemma** read-tmps-mono: \(\forall y s. \text{set } xs \subseteq \text{set } ys \implies \text{read-tmps } xs \subseteq \text{read-tmps } ys\)
by (fastforce simp add: in-read-tmps-conv)

**Fun** distinct-read-tmps:: 'p store-buffer ⇒ bool
where
  distinct-read-tmps [] = True
| distinct-read-tmps (r#rs) =
    (case r of
      Read\textsubscript{ab} volatile a t v ⇒ t \not∈ (read-tmps rs) \land \text{distinct-read-tmps } rs
| - ⇒ \text{distinct-read-tmps } rs)

**Lemma** distinct-read-tmps-conv:
distinct-read-tmps xs = (\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j −→
  (case xs!i of
    \text{Read}\textsubscript{ab} - - t_i - ⇒ case xs!j of \text{Read}\textsubscript{ab} - - t_j - ⇒ t_i \neq t_j | - ⇒ True
| - ⇒ True))
— Nice lemma, ugly proof.
**Proof** (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
show ?case
**Proof** (cases x)
case (Write\textsubscript{ab} volatile a sop v)
with Cons.hyps show ?thesis
  apply −
  apply (rule iffI [rule-format])
  apply clarsimp
    subgoal for i j
    apply (case-tac i)
    apply fastforce
    apply (case-tac j)
    apply (fastforce split: memref.splits)
    apply (clarsimp cong: memref.case-cong)
done
apply clarsimp
subgoal for i j
apply (erule-tac x=Suc i in allE)
apply clarsimp
apply (erule-tac x=Suc j in allE)
done

next
case (Read\textsubscript{ab} volatile a t v)
with Cons.hyps show ?thesis
apply −
apply (rule iffI [rule-format])
apply clarsimp
subgoal for i j
apply (case-tac i)
apply clarsimp
apply (case-tac j)
apply clarsimp
apply (fastforce split: memref.splits simp add: in-read-tmps-conv dest: nth-mem)
apply (clarsimp cong: memref.case-cong)
done
apply clarsimp
apply (rule conjI)
apply (clarsimp simp add: in-read-tmps-conv)
apply (erule-tac x=0 in allE)
apply (clarsimp simp add: in-set-conv-nth)
subgoal for volatile' a' v'i
apply (erule-tac x=Suc i in allE)
apply clarsimp
done

apply clarsimp
subgoal for i j
apply (erule-tac x=Suc i in allE)
apply clarsimp
apply clarsimp
apply (erule-tac x=Suc j in allE)
apply (clarsimp cong: memref.case-cong)
done
done

next
case Prog\textsubscript{ab}
with Cons.hyps show ?thesis
apply −
apply (rule iffI [rule-format])
apply clarsimp
subgoal for i j
apply (case-tac i)
apply fastforce
apply (case-tac j)
apply (fastforce split: memref.splits)
apply (clarsimp cong: memref.case-cong)
done

apply clarsimp

subgoal for i j
apply (erule-tac x=Suc i in allE)
apply clarsimp

apply (erule-tac x=Suc j in allE)
apply (clarsimp cong: memref.case-cong)
done
done

next

case Ghost_of

with Cons.hyps show ?thesis

apply –
apply (rule iffI [rule-format])
apply clarsimp

subgoal for i j
apply (case-tac i)
apply fastforce
apply (case-tac j)
apply (fastforce split: memref.splits)
apply (clarsimp cong: memref.case-cong)
done

apply clarsimp

subgoal for i j
apply (erule-tac x=Suc i in allE)
apply clarsimp

apply (erule-tac x=Suc j in allE)
apply (clarsimp cong: memref.case-cong)
done
done

qed

fun load-tmps:: instrs ⇒ tmp set

where
load-tmps [] = {}
| load-tmps (i#is) =
  (case i of
   Read volatile a t ⇒ insert t (load-tmps is)
   | RMW a t sop cond ret A L R W ⇒ insert t (load-tmps is)
   | _ ⇒ load-tmps is)

lemma in-load-tmps-conv:
(t ∈ load-tmps xs) = ((∃ volatile a. Read volatile a t ∈ set xs) ∨
(∃ a sop cond ret A L R W. RMW a t sop cond ret A L R W ∈ set xs))
by (induct xs) (auto split: instr.splits)

lemma load-tmps-mono: ∀ys. set xs ⊆ set ys ⇒ load-tmps xs ⊆ load-tmps ys
by (fastforce simp add: in-load-tmps-conv)
fun distinct-load-tmps:: instrs ⇒ bool

where

distinct-load-tmps [] = True
| distinct-load-tmps (r#rs) =
  (case r of
    Read volatile a t ⇒ t /∈ (load-tmps rs) ∧ distinct-load-tmps rs
  | RMW a t sop cond ret A L R W ⇒ t /∈ (load-tmps rs) ∧ distinct-load-tmps rs
  | - ⇒ distinct-load-tmps rs)

locale load-tmps-distinct =
fixes ts::('p, 'p store-buffer, bool, owns, rels) thread-config list
assumes load-tmps-distinct:
Σ i p is O R D δ sb.
[i < length ts; ts!i = (p,is,0, sb,D,O,R) ]
⇒
distinct-load-tmps is

locale read-tmps-distinct =
fixes ts::('p, 'p store-buffer, bool, owns, rels) thread-config list
assumes read-tmps-distinct:
Σ i p is O R D δ sb.
[i < length ts; ts!i = (p,is,0, sb,D,O,R) ]
⇒
distinct-read-tmps sb

locale load-tmps-read-tmps-distinct =
fixes ts::('p, 'p store-buffer, bool, owns, rels) thread-config list
assumes load-tmps-read-tmps-distinct:
Σ i p is O R D δ sb.
[i < length ts; ts!i = (p,is,0, sb,D,O,R) ]
⇒
load-tmps is ∩ read-tmps sb = {}

locale tmps-distinct =
load-tmps-distinct +
read-tmps-distinct +
load-tmps-read-tmps-distinct

lemma rev-read-tmps: read-tmps (rev xs) = read-tmps xs
by (auto simp add: in-read-tmps-conv)

lemma rev-load-tmps: load-tmps (rev xs) = load-tmps xs
by (auto simp add: in-load-tmps-conv)

lemma distinct-read-tmps-append: ∀ys. distinct-read-tmps (xs @ ys) =
(distinct-read-tmps xs ∧ distinct-read-tmps ys ∧
read-tmps xs ∩ read-tmps ys = {})
by (induct xs) (auto split: memref.splits simp add: in-read-tmps-conv)
lemma distinct-load-tmps-append: \( \forall ys. \) distinct-load-tmps \((xs @ ys) = (\text{distinct-load-tmps } xs \land \text{distinct-load-tmps } ys \land load-tmps \ xs \cap load-tmps \ ys = \{\}) \)
apply (induct xs)
apply (auto split: instr.splits simp add: in-load-tmps-conv)
done

lemma read-tmps-append: read-tmps \((xs@ys) = (\text{read-tmps } xs \cup \text{read-tmps } ys) \)
by (fastforce simp add: in-read-tmps-conv)

lemma load-tmps-append: load-tmps \((xs@ys) = (\text{load-tmps } xs \cup \text{load-tmps } ys) \)
by (fastforce simp add: in-load-tmps-conv)

fun write-sops:: \('p store-buffer \Rightarrow \) sop set
where
  write-sops [] = {}
| write-sops (r#rs) =
  (case r of
    Write sb volatile a sop v A L R W - - - - \Rightarrow insert sop (write-sops rs)
  | - \Rightarrow write-sops rs)

lemma in-write-sops-conv:
  \((sop \in \text{write-sops } xs) = (\exists \text{volatile } a \ \text{v A L R W. Write}_{sb} \ \text{volatile } a \ \text{sop v A L R W } \in \text{set } xs)) \)
apply (induct xs)
apply simp
apply (auto split: memref.splits)
apply force
apply force
done

lemma write-sops-mono:
  \( \forall ys. \) set \( x \subseteq \text{set } y \implies \text{write-sops } x \subseteq \text{write-sops } y \)
by (fastforce simp add: in-write-sops-conv)

lemma write-sops-append: write-sops \((xs@ys) = \text{write-sops } xs \cup \text{write-sops } ys \)
by (force simp add: in-write-sops-conv)

fun store-sops:: instrs \Rightarrow sop set
where
  store-sops [] = {}
| store-sops (i#is) =
  (case i of
    Write volatile a sop v A L R W - - - - \Rightarrow insert sop (store-sops is)
  | RMW a t sop cond ret A L R W - \Rightarrow insert sop (store-sops is)
  | - \Rightarrow store-sops is)

lemma in-store-sops-conv:
(sop \in\text{store-sops}\ xs) = ((\exists\text{volatile}\ a\ A\ L\ R\ W.\ \text{Write}\ \text{volatile}\ a\ A\ L\ R\ W\ \in\ \text{set}\ xs)\\\lor\\(\exists\ a\ t\ \text{cond}\ \text{ret}\ A\ L\ R\ W.\\ RMW\ a\ t\ \text{cond}\ \text{ret}\ A\ L\ R\ W\ \in\ \text{set}\ xs))\\by\ (\text{induct}\ xs)\ (\text{auto}\ \text{split}:\ \text{instr.splits})

\textbf{lemma} \text{store-sops-mono}: \bigwedge ys.\ \text{set}\ xs \subseteq\ \text{set}\ ys \implies\ \text{store-sops}\ xs \subseteq\ \text{store-sops}\ ys\\by\ (\text{fastforce}\ \text{simp}\ \text{add}:\ \text{in-store-sops-conv})

\textbf{lemma} \text{store-sops-append}: \text{store-sops}\ (xs@ys) = \text{store-sops}\ xs \cup\ \text{store-sops}\ ys\\by\ (\text{force}\ \text{simp}\ \text{add}:\ \text{in-store-sops-conv})

\text{locale} \text{valid-write-sops} = 
\text{fixes} ts::(p', p\ \text{store-buffer}, bool, owns, rels) thread-config list
\textbf{assumes} \text{valid-write-sops}:
\bigwedge i\ p\ \text{is}\ \mathcal{O}\ \mathcal{R}\ \mathcal{D}\ \emptyset\ \text{sb}.
[i < \text{length}\ ts;\ ts!i = (p, is, \emptyset, sb, \mathcal{O}, \mathcal{R})]\\implies\\\forall\ \text{sop} \in\ \text{write-sops}\ \text{sb}.\ \text{valid-sop}\ \text{sop}

\text{locale} \text{valid-store-sops} = 
\text{fixes} ts::(p', p\ \text{store-buffer}, bool, owns, rels) thread-config list
\textbf{assumes} \text{valid-store-sops}:
\bigwedge i\ \text{is}\ \mathcal{O}\ \mathcal{R}\ \mathcal{D}\ \emptyset\ \text{sb}.
[i < \text{length}\ ts;\ ts!i = (p, is, \emptyset, sb, \mathcal{O}, \mathcal{R})]\\implies\\\forall\ \text{sop} \in\ \text{store-sops}\ \text{is}.\ \text{valid-sop}\ \text{sop}

\text{locale} \text{valid-sops} = \text{valid-write-sops} + \text{valid-store-sops}

The value stored in a non-volatile \text{Read}_{sb} in the store-buffer has to match the last value written to the same address in the store buffer or the memory content if there is no corresponding write in the store buffer. No volatile read may follow a volatile write. Volatile reads in the store buffer may refer to a stale value: e.g. imagine one writer and multiple readers \textbf{fun} \text{reads-consistent}:: bool \Rightarrow\ owns \Rightarrow\ memory \Rightarrow p' \text{store-buffer} \Rightarrow bool
\textbf{where}
\begin{align*}
\text{reads-consistent pending-write} &\mathcal{O}\ m\ [] = True \\
|\ \text{reads-consistent pending-write} &\mathcal{O}\ m\ (r#rs) = \begin{cases}
\text{case}\ r\ \text{of} \\
\text{Read}_{sb}\ \text{volatile}\ a\ t\ v &\Rightarrow (\neg\ \text{volatile} \rightarrow (\text{pending-write} \lor a \in \mathcal{O}) \rightarrow v = m\ a) \land \\
&\ \text{reads-consistent pending-write} \ O\ m\ rs \\
|\ \text{Write}_{sb}\ \text{volatile}\ a\ \text{ sop}\ v\ A\ L\ R\ W &\Rightarrow \\
&\ (\text{if}\ \text{volatile} \text{then} \\
&\ \ \ \ \ \text{outstanding-refs is-volatile-Read}_{sb}\ rs = \{} \land \\
&\ \ \ \ \ \ \text{reads-consistent True} (\mathcal{O} \cup A - R) (m(a := v))\ rs \\
&\ \ \ \ \ \ \ \text{else}\ \text{reads-consistent pending-write} \mathcal{O} (m(a := v))\ rs)
\end{cases}
| Ghost_{sb} A L R W &\Rightarrow\ \text{reads-consistent pending-write} (\mathcal{O} \cup A - R) m\ rs \\
| - &\Rightarrow\ \text{reads-consistent pending-write} \mathcal{O} m\ rs
\end{align*}

\textbf{fun} \text{volatile-reads-consistent}:: memory \Rightarrow p' \text{store-buffer} \Rightarrow bool
\textbf{where
volatile-reads-consistent m [] = True
| volatile-reads-consistent m (r#rs) =
  (case r of
   Read sb volatile a t v ⇒ (volatile → v = m a) ∧ volatile-reads-consistent m rs
   | Write sb volatile a sop v A L R W ⇒ volatile-reads-consistent (m(a := v)) rs
   | - ⇒ volatile-reads-consistent m rs
  )

fun flush:: 'p store-buffer ⇒ memory ⇒ memory
where
  flush [] m = m
  | flush (r#rs) m =
    (case r of
     Write sb volatile a - v - - - - ⇒ flush rs (m(a:=v))
     | - ⇒ flush rs m)

lemma reads-consistent-pending-write-antimono:
  ∀ O m. reads-consistent True O m sb ⇒ reads-consistent False O m sb
apply (induct sb)
apply simp
subgoal for a sb O m
  by (case-tac a) auto
done

lemma reads-consistent-owns-antimono:
  ∀ O' pending-write m.
    O ⊆ O' ⇒ reads-consistent pending-write O' m sb ⇒ reads-consistent pending-write O m sb
apply (induct sb)
apply simp
subgoal for a sb O' pending-write m
apply (case-tac a)
apply (clar simp split: if-split-asm)
subgoal for volatile a D f v A L R W
apply (drule-tac C=A in union-mono-aux)
apply (drule-tac C=R in set-minus-mono-aux)
apply blast
done
apply fastforce
apply fastforce
apply clar simp
subgoal for A L R W
apply (drule-tac C=A in union-mono-aux)
apply (drule-tac C=R in set-minus-mono-aux)
apply blast
done
done
done
lemma acquired-reads-mono': x ∈ acquired-reads b xs A ⟷ acquired-reads b xs B = {}
⇒ A ⊆ B ⟷ False
apply (drule acquired-reads-mono-in [where B=B])
apply auto
done

lemma reads-consistent-append:
\(\forall m\) pending-write \(O\), reads-consistent pending-write \(O\) m (xs@ys) =
(\text{reads-consistent pending-write } O\ m\ \text{xs } \land
\text{reads-consistent (pending-write } \lor \text{ outstanding-refs is-volatile-Write}_{sb}\ \text{xs } \neq \{\})
\text{acquired True } \text{xs } O\ \text{flush } \text{xs } \text{m}\ \text{ys } \land
\text{outstanding-refs is-volatile-Write}_{sb}\ \text{xs } \neq \{\}
\rightarrow \text{outstanding-refs is-volatile-Read}_{sb}\ \text{ys } \neq \{\})
apply (induct xs)
apply clarsimp
subgoal for a xs m pending-write O
apply (case-tac a)
apply (auto simp add: outstanding-refs-append acquired-reads-append
dest: acquired-reads-mono-in acquired-pending-write-mono-in acquired-reads-mono'
acquired-mono-in)
done
done

lemma reads-consistent-mem-eq-on-non-volatile-reads:
assumes mem-eq: \(\forall a \in A\). m' a = m a
assumes subset: outstanding-refs is-non-volatile-Read_{sb} \(sb\) ⊆ A
— We could be even more restrictive here, only the non volatile reads that are not
buffered in \(sb\) have to be the same.
assumes consis-m: reads-consistent pending-write \(O\) m sb
shows reads-consistent pending-write \(O\) m' sb
using mem-eq subset consis-m
proof (induct sb arbitrary: m' m pending-write O)
case Nil thus ?case by simp
next
case (Cons r sb)
note mem-eq = (\forall a \in A\). m' a = m a
note subset = \{outstanding-refs is-non-volatile-Read_{sb}\ \(r\#sb\) \subseteq A\}
note consis-m = \{reads-consistent pending-write \(O\) m \(r\#sb\)\}

from subset have subset': outstanding-refs is-non-volatile-Read_{sb} \(sb\) \subseteq A
by (auto simp add: Write_{sb})
show ?case
proof (cases r)
case (Write_{sb} volatile a sop v A' L R W)
from mem-eq
have mem-eq'):
\(\forall a' \in A\). (m'(a:=v)) a' = (m(a:=v)) a'
by (auto)
show \( ?\)thesis
proof (cases volatile)
case True
from consis-m obtain
consis': reads-consistent True \((O \cup A' - R) (m(a := v))\) sb and
no-volatile-Read_{ab}: outstanding-refs is-volatile-Read_{ab} sb = {}
by (simp add: Write_{ab} True)

from Cons.hyps [OF mem-eq' subset' consis]
have reads-consistent True \((O \cup A' - R) (m'(a := v))\) sb.
with no-volatile-Read_{ab}
show ?thesis
by (simp add: Write_{ab} True)
next
case False
from consis-m obtain consis': reads-consistent pending-write O \((m(a := v))\) sb
by (simp add: Write_{ab} False)
from Cons.hyps [OF mem-eq' subset' consis]
have reads-consistent pending-write O \((m'(a := v))\) sb.
then
show ?thesis
by (simp add: Write_{ab} False)
qed
next
case \((Read_{ab} \text{ volatile } a \text{ t } v)\)
from mem-eq
have mem-eq':
\(\forall a' \in A. \ m' a' = m a'\)
by (auto)
show ?thesis
proof (cases volatile)
case True
from consis-m obtain
consis': reads-consistent pending-write O m sb
by (simp add: Read_{ab} True)
from Cons.hyps [OF mem-eq' subset' consis]
show ?thesis
by (simp add: Read_{ab} True)
next
case False
from consis-m obtain
consis': reads-consistent pending-write O m sb \text{ and } v: (pending-write \lor a \in O) \longrightarrow v=m a
by (simp add: Read_{ab} False)
from mem-eq subset Read_{ab} have m' a = m a
by (auto simp add: False)
with Cons.hyps [OF mem-eq' subset' consis'] v
show ?thesis
by (simp add: Read_{ab} False)
qed
next
case Prog$_{ab}$ with Cons show ?thesis by auto
next
case Ghost$_{sb}$ with Cons show ?thesis by auto
qed
qed

lemma volatile-reads-consistent-mem-eq-on-volatile-reads:
assumes mem-eq: $\forall a \in A. \ m' a = m a$
assumes subset: outstanding-refs (is-volatile-Read$_{ab}$) sb $\subseteq A$
— We could be even more restrictive here, only the non volatile reads that are not
buffered in sb have to be the same.
assumes consis-m: volatile-reads-consistent m sb
shows volatile-reads-consistent m' sb
using mem-eq subset consis-m
proof (induct sb arbitrary: m' m)
case Nil thus ?case by simp
next
case (Cons r sb)
note mem-eq = $\langle \forall a \in A. \ m' a = m a \rangle$
note subset = $\langle$outstanding-refs (is-volatile-Read$_{ab}$) (r#sb) $\subseteq A$\rangle
note consis-m = $\langle$volatile-reads-consistent m (r#sb)$\rangle$

from subset have subset': outstanding-refs is-volatile-Read$_{ab}$ sb $\subseteq A$
by (auto simp add: Write$_{ab}$)
show ?case
proof (cases r)
case (Write$_{ab}$ volatile a sop v A' L R W)
from mem-eq
have mem-eq':
$\forall a' \in A. \ (m'(a:=v)) a' = (m(a:=v)) a'$
by (auto)
show ?thesis
proof (cases volatile)
case True
from consis-m obtain
consis': volatile-reads-consistent (m(a := v)) sb
by (simp add: Write$_{ab}$ True)

from Cons.hyps [OF mem-eq' subset' consis']
have volatile-reads-consistent (m'(a := v)) sb.
then
show ?thesis
by (simp add: Write$_{ab}$ True)
next
case False
from consis-m obtain consis': volatile-reads-consistent (m(a := v)) sb
by (simp add: Write$_{ab}$ False)
from Cons.hyps [OF mem-eq’ subset’ consis’]
have volatile-reads-consistent (m'(a := v)) sb.
then
  show ?thesis
by (simp add: Write\sb False)
qed

next
  case (Read\sb volatile a t v)
  from mem-eq
  have mem-eq':
    \(\forall a' \in A. \ m'a' = m\ a'\)
by (auto)
  show ?thesis
proof (cases volatile)
  case False
  from consis-m obtain
  consis': volatile-reads-consistent m sb
by (simp add: Read\sb False)
from Cons.hyps [OF mem-eq’ subset’ consis’]
show ?thesis
by (simp add: Read\sb False)
next
  case True
  from consis-m obtain
  consis': volatile-reads-consistent m sb and v: v = m a
by (simp add: Read\sb True)
from mem-eq subset Read\sb v have v = m a
by (auto simp add: True)
with Cons.hyps [OF mem-eq’ subset’ consis’]
show ?thesis
by (simp add: Read\sb True)
qed

next
  case Prog\sb with Cons
  show ?thesis
with auto
next
  case Ghost\sb with Cons
  show ?thesis
with auto
qed

locale valid-reads =
fixes m::memory and ts::('p, 'p store-buffer,bool,owns,rels) thread-config list
assumes valid-reads: \(\forall i \ p \ is \ \mathcal{O} \mathcal{R} \mathcal{D} \ \emptyset \ sb.\)
  \[ [ i < \text{length} \ ts; \ ts!i = (p, is, \emptyset, sb, D,\mathcal{O},\mathcal{R}) ] \implies \)
  reads-consistent False \mathcal{O} m sb

lemma valid-reads-Cons: valid-reads m (t\#ts) =
(\let (--,--\sb,--\mathcal{O},-) = t \text{ in reads-consistent False } \mathcal{O} m sb \land \text{valid-reads m ts})
apply (auto simp add: valid-reads-def)
subgoal for p' is' \emptyset sb' D' \mathcal{O}' \mathcal{R}' i \ p \ is \ \emptyset \ sb \ D \ \mathcal{O} \ \mathcal{R}
apply (case-tac i)

178
apply auto
done
done

Readsb5 and writes have in the store-buffer have to conform to the valuation of temporaries. context program
begin
fun history-consistent:: tmps ⇒ 'p ⇒ 'p store-buffer ⇒ bool
where
  history-consistent θ p [] = True
| history-consistent θ p (r#rs) =
  (case r of
    Readsb vol a t v ⇒
      (case θ t of Some v' ⇒ v=v' ∧ history-consistent θ p rs | - ⇒ False)
| Write sb vol a (D,f) v - - - - ⇒
  D ⊆ dom θ ∧ f θ = v ∧ D ∩ read-tmps rs = {} ∧ history-consistent θ p rs
| Progsb p1 p2 is ⇒ p1=p ∧
  θ|(dom θ - read-tmps rs) ⊢ p1 →p (p2,is) ∧
  history-consistent θ p2 rs
| - ⇒ history-consistent θ p rs)
end

fun hd-prog:: 'p ⇒ 'p store-buffer ⇒ 'p
where
  hd-prog p [] = p
| hd-prog p (i#is) = (case i of
    Progsb p' - - ⇒ p'
| - ⇒ hd-prog p is)

fun last-prog:: 'p ⇒ 'p store-buffer ⇒ 'p
where
  last-prog p [] = p
| last-prog p (i#is) = (case i of
    Progsb - p' - ⇒ last-prog p' is
| - ⇒ last-prog p is)

locale valid-history = program +
constrains
  program-step :: tmps ⇒ 'p ⇒ 'p × instrs ⇒ bool
fixes ts::('(p,p store-buffer,bool,owns,rels) thread-config list
assumes valid-history: \i p is O R D θ sb.
  [i < length ts; ts!i = (p,is,θ,db,D,O,R) ] ⇒
  program.history-consistent program-step θ (hd-prog p sb) sb

fun data-dependency-consistent-instrs:: addr set ⇒ instrs ⇒ bool
where
  data-dependency-consistent-instrs T [] = True
| data-dependency-consistent-instrs T (i#is) =
  (case i of
    Write volatile a (D,f) - - - - ⇒ D ⊆ T ∧ D ∩ load-tmps is = {} ∧
    data-dependency-consistent-instrs T is

179
lemma data-dependency-consistent-mono:
\[ T \subseteq T' \quad \text{if} \quad \text{data-dependency-consistent-instrs } T \quad \text{is}; \quad T \subseteq T' \]
\[ \Rightarrow \]
\[ \text{data-dependency-consistent-instrs } T' \quad \text{is} \]
apply (induct is)
apply clasimp
subgoal for a is T T'
apply (case-tac a)
apply clasimp
  subgoal for volatile a' t
  apply (drule-tac a=t in insert-mono)
  apply clasimp
done
apply fastforce
apply clasimp
  subgoal for a' t D f cond ret A L R W
  apply (frule-tac a=t in insert-mono)
  apply fastforce
done
apply fastforce
done
done

lemma data-dependency-consistent-instrs-append:
\[ \forall ys T. \quad \text{data-dependency-consistent-instrs } T \quad \text{(xs@ys)} = \]
\[ \text{(data-dependency-consistent-instrs } T \quad \text{xs} \quad \land \]
\[ \text{data-dependency-consistent-instrs } (T \cup \text{load-tmps xs}) \quad ys \quad \land \]
\[ \text{load-tmps ys} \quad \cap \quad \bigcup \quad (\text{fst ' store-sops xs}) = \{\} \]
apply (induct xs)
apply (auto split: instr.splits simp add: load-tmps-append intro: data-dependency-consistent-mono)
done

locale valid-data-dependency =
fixes ts::('p, 'p store-buffer, bool, owns, rels) thread-config list
assumes data-dependency-consistent-instrs:
\[ \forall i \quad \text{p is } O \quad D \quad \emptyset \quad sb. \]
\[ [i < \text{length ts}; \quad ts!i = (p, \text{is, } \emptyset \text{, sb, D, O, R}) \quad ] \quad \Rightarrow \]
\[ \text{data-dependency-consistent-instrs } (\text{dom } \emptyset) \quad \text{is} \]
assumes load-tmps-write-tmps-distinct:
\[ \forall i \quad \text{p is } O \quad D \quad \emptyset \quad sb. \]
\[ [i < \text{length ts}; \quad ts!i = (p, \text{is, } \emptyset \text{, sb, D, O, R}) \quad ] \quad \Rightarrow \]
\[ \text{load-tmps is } \cap \quad \bigcup \quad (\text{fst ' write-sops sb}) = \{\} \]
locale load-tmps-fresh =

fixes ts::(p, p store-buffer, bool, owns, rels) thread-config list

assumes load-tmps-fresh:
\[ i.p is D.D \emptyset sb. \]
\[ [i < length ts; ts!i = (p, is, D.D, D.R)] \implies 
load-tmps is \cap dom \emptyset = \{\} \]

fun acquired-by-instrs :: instrs ⇒ addr set ⇒ addr set
where
acquired-by-instrs [] A = A
| acquired-by-instrs (i#is) A =
  (case i of
   Read volatile a - - \⇒ acquired-by-instrs is A
   | Write volatile a - - A' L R W ⇒ acquired-by-instrs is (if volatile then (A U A' - R)
else A)
   | RMW a t sop cond ret A' L R W ⇒ acquired-by-instrs is {} 
   | Fence ⇒ acquired-by-instrs is {} 
   | Ghost A' L R W ⇒ acquired-by-instrs is (A U A' - R)) 

fun acquired-loads :: bool ⇒ instrs ⇒ addr set ⇒ addr set
where
acquired-loads pending-write [] A = {}
| acquired-loads pending-write (i#is) A =
  (case i of
   Read volatile a - - ⇒ (if pending-write \land \lnot volatile \land a \in A
   then insert a (acquired-loads pending-write is A)
   else acquired-loads pending-write is A)
   | Write volatile a - - A' L R W ⇒ (if volatile then acquired-loads True is (if pending-write
then (A U A' - R) else {}})
   else acquired-loads pending-write is A)
   | RMW a t sop cond ret A' L R W ⇒ acquired-loads pending-write is {}
   | Fence ⇒ acquired-loads pending-write is {} 
   | Ghost A' L R W ⇒ acquired-loads pending-write is (A U A' - R)) 

lemma acquired-by-instrs-mono:
\[ \land A. B. A \subseteq B \implies acquired-by-instrs is A \subseteq acquired-by-instrs is B \]
apply (induct is)
apply simp
subgoal for a is A B
apply (case-tac a)
apply clarsimp
apply clarsimp
apply clarsimp
subgoal for volatile a' D f A' L R W x
apply (drule-tac C=A' in union-mono-aux)
apply (drule-tac C=R in set-minus-mono-aux)
apply blast
done
apply clarsimp
apply clarsimp
apply clarsimp
subgoal for A’ L R W x
apply (drule-tac C=A’ in union-mono-aux)
apply (drule-tac C=R in set-minus-mono-aux)
apply blast
done
done
done

lemma acquired-by-instrs-mono-in:
  assumes x-in: x ∈ acquired-by-instrs is A
  assumes sub: A ⊆ B
  shows x ∈ acquired-by-instrs is B
using acquired-by-instrs-mono [OF sub, of is] x-in
by blast

locale enough-flushs =
fixes ts::('p, 'p store-buffer, bool, owns, rels) thread-config list
assumes clean-no-outstanding-volatile-Write sb:
    ⋀ i p is ORD θ sb.
    [i < length ts; ts!i = (p, is, D, O, R); ¬D]
    ⟹ (outstanding-refs is-volatile-Write sb sb = {})

fun prog-instrs:: 'p store-buffer ⇒ instrs
where
  prog-instrs [] = []
  | prog-instrs (i#is) = (case i of
    Prog sb - - is' ⇒ is' @ prog-instrs is
    | - ⇒ prog-instrs is)

fun instrs:: 'p store-buffer ⇒ instrs
where
  instrs [] = []
  | instrs (i#is) = (case i of
    Write sb volatile a sop v A L R W ⇒ Write volatile a sop A L R W# instrs is
    | Read sb volatile a t v ⇒ Read volatile a t # instrs is
    | Ghost sb A L R W ⇒ Ghost A L R W# instrs is
    | - ⇒ instrs is)

locale causal-program-history =
fixes is sb and sb
assumes causal-program-history:
    ⋀ sb1 sb2. sb=sb1@sb2 ⟹ ∃ is. instrs sb2 @ is sb = is @ prog-instrs sb2

lemma causal-program-history-empty [simp]: causal-program-history is []
by (rule causal-program-history-empty.intro) simp

lemma causal-program-history-suffix:
  causal-program-history is sb (sb@sb') ⟹ causal-program-history is sb sb'
by (auto simp add: causal-program-history-def)
locale valid-program-history =
  fixes ts::('p, p store-buffer, bool, owns, rels) thread-config list
assumes valid-program-history:
  \( \forall i \ p \exists O \ R \ D \ p \ \text{is} \ \text{valid-program-history} \)
  \[ i < \text{length ts}; ts[i] = (p, is, \theta, sb, D, O, R) \] \Rightarrow
  causal-program-history is sb
assumes valid-last-prog:
  \( \forall i \ p \exists O \ R \ D \ p \ \text{is} \ \text{valid-last-prog} \)
  \[ i < \text{length ts}; ts[i] = (p, is, \theta, sb, D, O, R) \] \Rightarrow
  last-prog p sb = p

lemma (in valid-program-history) valid-program-history-nth-update:
  \[ [i < \text{length ts}; \text{causal-program-history is sb}; \text{last-prog p sb = p}] \]
  \Rightarrow
  valid-program-history (ts [i := (p, is, \theta, sb, D, O, R)])
  by (rule valid-program-history.intro)
  (auto dest: valid-program-history valid-last-prog
   simp add: nth-list-update split: if-split-asm)

lemma (in outstanding-non-volatile-refs-owned-or-read-only)
  outstanding-non-volatile-refs-owned-instructions-read-value-independent:
  \( \forall i \ p \exists O \ R \ D \ p \ \text{is} \ \text{outstanding-non-volatile-refs-owned-or-read-only} \)
  \[ i < \text{length ts}; ts[i] = (p, is, \theta, sb, D, O, R) \] \Rightarrow
  outstanding-non-volatile-refs-owned-or-read-only S (ts[i := (p', is', \theta', sb, D', O, R)])
  by (unfold-locales)
  (auto dest: outstanding-non-volatile-refs-owned-or-read-only
   simp add: nth-list-update split: if-split-asm)

lemma (in outstanding-non-volatile-refs-owned-or-read-only)
  outstanding-non-volatile-refs-owned-or-read-only-nth-update:
  \( \forall i \ p \exists O \ R \ D \ p \ \text{is} \ \text{outstanding-non-volatile-refs-owned-or-read-only} \)
  \[ i < \text{length ts}; \text{non-volatile-owned-or-read-only False S O sb} \] \Rightarrow
  outstanding-non-volatile-refs-owned-or-read-only S (ts[i := (p, is, \theta, sb, D, O, R)])
  by (unfold-locales)
  (auto dest: outstanding-non-volatile-refs-owned-or-read-only
   simp add: nth-list-update split: if-split-asm)

lemma (in outstanding-volatile-writes-unowned-by-others)
  outstanding-volatile-writes-unowned-by-others-instructions-read-value-independent:
  \( \forall i \ p \exists O \ R \ D \ p \ \text{is} \ \text{outstanding-volatile-writes-unowned-by-others} \)
  \[ i < \text{length ts}; ts[i] = (p, is, \theta, sb, D, O, R) \] \Rightarrow
  outstanding-volatile-writes-unowned-by-others (ts[i := (p', is', \theta', sb, D', O, R)])
  by (unfold-locales)
  (auto dest: outstanding-volatile-writes-unowned-by-others
   simp add: nth-list-update split: if-split-asm)

lemma (in read-only-reads-unowned)
  read-only-unowned-instructions-read-value-independent:
\( i \) is \( \mathcal{O} \) \( \mathcal{R} \) \( \mathcal{D} \) \( \notin \) \( \text{sb} \).
[i < length ts; ts!i = (p,is,\theta,\text{sb},D,O,R)] \implies \text{read-only-reads-unowned (ts[i := (p,is',\theta',\text{sb},D',O,R'])]}

by (unfold-locales)
(auto dest: read-only-reads-unowned
simp add: nth-list-update split: if-split-asm)

lemma Write\(_{sb}\)-in-outstanding-refs:
Write\(_{sb}\) True a sop v A L R W \in set xs \implies a \in outstanding-refs is-volatile-Write\(_{sb}\) xs
by (induct xs) (auto split:memref.splits)

lemma (in outstanding-volatile-writes-unowned-by-others)
outstanding-volatile-writes-unowned-by-others-store-buffer:
\( \forall i \) p is \( \mathcal{O} \) \( \mathcal{R} \) \( \mathcal{D} \) \( \notin \) \( \text{sb} \).
[i < length ts; ts!i = (p,is,\theta,\text{sb},D,O,R);
outstanding-refs is-volatile-Write\(_{sb}\) \( \text{sb}' \subseteq \) outstanding-refs is-volatile-Write\(_{sb}\) \( \text{sb} \);
all-acquired \( \text{sb}' \subseteq \) all-acquired \( \text{sb} \)] \implies
outstanding-volatile-writes-unowned-by-others (ts[i := (p,is',\theta',\text{sb}',D',O,R')])
apply (unfold-locales)
apply (fastforce dest: outstanding-volatile-writes-unowned-by-others
simp add: nth-list-update split: if-split-asm)
done

lemma (in ownership-distinct)
ownership-distinct-instructions-read-value-store-buffer-independent:
\( \forall i \) p is \( \mathcal{O} \) \( \mathcal{R} \) \( \mathcal{D} \) \( \notin \) \( \text{sb} \).
[i < length ts; ts!i = (p,is,\theta,\text{sb},D,O,R);
all-acquired \( \text{sb}' \subseteq \) all-acquired \( \text{sb} \)] \implies
ownership-distinct (ts[i := (p,is',\theta',\text{sb}',D',O,R')])
by (unfold-locales)
(auto dest: ownership-distinct
simp add: nth-list-update split: if-split-asm)

lemma (in ownership-distinct)
ownership-distinct-nth-update:
\( \forall i \) p is \( \mathcal{O} \) \( \mathcal{R} \) \( \mathcal{D} \) \( \text{xs} \) \( \text{sb} \).
[i < length ts; ts!i = (p,is,\theta,\text{sb},D,O,R);
\( \forall j \) j \( \neq j \) \( \implies \) (let (p\(_j\),is\(_j\),\theta\(_j\),\text{sb}\(_j\),D\(_j\),O\(_j\),R\(_j\)) = ts!j in (O\(_j\) \cup \text{all-acquired} \( \text{sb}' \)) \cap (O\(_j\) \cup \text{all-acquired} \( \text{sb}\(_j\))) = \{\})] \implies
ownership-distinct (ts[i := (p,is',\theta',\text{sb}',D',O',R')])
apply (unfold-locales)
apply (clarsimp simp add: nth-list-update split: if-split-asm)
apply (force dest: ownership-distinct simp add: Let-def)
apply (fastforce dest: ownership-distinct simp add: Let-def)
apply (fastforce dest: ownership-distinct simp add: Let-def)
lemma (in valid-write-sops) valid-write-sops-nth-update:
  \[ i < \text{length } ts; \forall \text{sop } \in \text{write-sops } sb. \text{ valid-sop sop} \] \implies 
  \text{valid-write-sops } (ts[i := (p, is, xs, sb, D, O, R)])
by (unfold valid-write-sops-def)
(auto dest: valid-write-sops simp add: nth-list-update split: if-split-asm)

lemma (in valid-store-sops) valid-store-sops-nth-update:
  \[ i < \text{length } ts; \forall \text{sop } \in \text{store-sops } is. \text{ valid-sop sop} \] \implies 
  \text{valid-store-sops } (ts[i := (p, is, xs, sb, D, O, R)])
by (unfold valid-store-sops-def)
(auto dest: valid-store-sops simp add: nth-list-update split: if-split-asm)

lemma (in valid-sops) valid-sops-nth-update:
  \[ i < \text{length } ts; \forall \text{sop } \in \text{write-sops } sb. \text{ valid-sop sop}; \forall \text{sop } \in \text{store-sops } is. \text{ valid-sop sop} \] \implies 
  \text{valid-sops } (ts[i := (p, is, xs, sb, D, O, R)])
by (unfold valid-sops-def valid-write-sops-def valid-store-sops-def)
(auto dest: valid-write-sops valid-store-sops simp add: nth-list-update split: if-split-asm)

lemma (in valid-data-dependency) valid-data-dependency-nth-update:
  \[ i < \text{length } ts; \text{ data-dependency-consistent-instrs } (\text{dom } \theta) \text{ is}; \text{ load-tmps } is \cap \bigcup (\text{fst } \text{write-sops } sb) = \{\} \] \implies 
  \text{valid-data-dependency } (ts[i := (p, is, \theta, sb, D, O, R)])
by (unfold valid-data-dependency-def)

lemma (in enough-flushs) enough-flushs-nth-update:
  \[ i < \text{length } ts; \neg D \longrightarrow (\text{outstanding-ref is-volatile-Write}_sb \text{ sb = } \{\}) \] \implies 
  \text{enough-flushs } (ts[i := (p, is, \theta, sb, D, O, R)])
apply (unfold-locales)
  apply (force simp add: nth-list-update split: if-split-asm dest: clean-no-outstanding-volatile-Write\_sb)
done

lemma (in outstanding-non-volatile-writes-unshared)
outstanding-non-volatile-writes-unshared-nth-update:
  \[ i < \text{length } ts; \text{ non-volatile-writes-unshared } S \text{ sb} \] \implies 
  \text{outstanding-non-volatile-writes-unshared } S (ts[i := (p, is, xs, sb, D, O, R)])
by (unfold-locales)
(auto dest: outstanding-non-volatile-writes-unshared simp add: nth-list-update split: if-split-asm)
lemma (in sharing-consistent)
sharing-consis-nth-update:
[i < length ts; sharing-consistent S O sb] \implies
sharing-consis S (ts[i := (p,is,xs,sb,D,O,R)])
by (unfold-locales)
(auto dest: sharing-consis
simp add: nth-list-update split: if-split-asm)

lemma (in no-outstanding-write-to-read-only-memory)
o-outstanding-write-to-read-only-memory-nth-update:
[i < length ts; no-write-to-read-only-memory S sb] \implies
no-outstanding-write-to-read-only-memory S (ts[i := (p,is,xs,sb,D,O,R)])
by (unfold-locales)
(auto dest: no-outstanding-write-to-read-only-memory
simp add: nth-list-update split: if-split-asm)

lemma in-Union-image-nth-conv: a \in \bigcup (f ' set xs) \implies \exists i. i < length xs \land a \in f (xs!i)
by (auto simp add: in-set-conv-nth)

lemma in-Inter-image-nth-conv: a \in \bigcap (f ' set xs) = (\forall i < length xs. a \in f (xs!i))
by (force simp add: in-set-conv-nth)

lemma release-ownership-nth-update:
assumes R-subset: R \subseteq O
shows \( \bigwedge i. [i < length ts; ts!i = (p,is,xs,sb,D,O,R); ownership-distinct ts] \implies \bigcup (\lambda (-,-,-,-,O,-) . O) ' set (ts[i:=(p,is',xs',sb',D',O - R,R)]) = ((\bigcup ((\lambda(-,-,-,-,O,-) . O) ' set ts)) - R) \)
proof (induct ts)
case Nil thus ?case by simp
next
case (Cons t ts)
note i-bound = \langle i < length (t # ts) \rangle
note ith = \langle (t # ts) ! i = (p,is,xs,sb,D,O,R) \rangle
note dist = \langle ownership-distinct (t#ts) \rangle
then interpret ownership-distinct t#ts.
from dist
have dist': ownership-distinct ts
  by (rule ownership-distinct-tl)
show ?case
proof (cases i)
case 0
from ith 0 have t: t = (p,is,xs,sb,D,O,R)
  by simp
have R \cap (\bigcup ((\lambda(-,-,-,-,O,-) . O) ' set ts)) = {}
proof -
{
fix x
assume x-R: x ∈ R
assume x-ls: x ∈ (∪((λ(−,−,−,−,−,O,−). O) i set ts))
then obtain j p_j i_j O_j R_j D_j x_j s_b_j where
  j-bound: j < length ts and
  jth: ts!j = (p_j,i_j,x_j,s_b_j,D_j,O_j,R_j) and
  x-in: x ∈ O_j
  by (fastforce simp add: in-set-conv-nth)
from j-bound jth 0
have (O ∪ all-acquired s_b) ∩ (O_j ∪ all-acquired s_b_j) = {}
  apply -
  apply (rule ownership-distinct [OF i-bound - - ith, of Suc j])
  apply clarsimp+
  apply blast
  done

with x-R R-subset x-in have False
  by auto
  }
  thus ?thesis
by blast
qed
then
show ?thesis
  by (auto simp add: 0 t)
next
  case (Suc n)
  obtain p_l i_l O_l R_l D_l x_l s_b_l where t: t = (p_l,i_l,x_l,s_b_l,D_l,O_l,R_l)
    by (cases t)
  have n-bound: n < length ts
    using i-bound by (simp add: Suc)
  have nth: ts!n = (p,i_,x_,s_b,D,O,R)
    using ith by (simp add: Suc)
  have R ∩ (O_l ∪ all-acquired s_b_l) = {}
  proof -
  { }
  fix x
  assume x-R: x ∈ R
  assume x-owns_l: x ∈ (O_l ∪ all-acquired s_b_l)
  from t
  have (O ∪ all-acquired s_b) ∩ (O_l ∪ all-acquired s_b_l) = {}
    apply -
    apply (rule ownership-distinct [OF i-bound - - ith, of 0])
    apply (auto simp add: Suc)
    done
  with x-owns_l x-R R-subset have False
by auto

thus thesis

by blast

qed

with Cons.hyps [OF n-bound nth dist']

show thesis

by (auto simp add: Suc t)

qed

lemma acquire-ownership-nth-update:

shows \( \forall i. \ [i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, R)] \)

\[ \implies \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot, O). \ O) \setminus \text{set} \ (ts[i:=(p',is',xs',sb',D',O \cup A,R')])) \]

= \((\bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot, O). \ O) \setminus \text{set} \ ts)) \cup A \)

proof (induct ts)

next

\[ \begin{align*}
\text{case Nil} & \quad \text{thus } \text{?case by simp} \\
\text{case (Cons t ts)} & \quad \text{next} \\
\text{note i-bound } = & \quad i < \text{length (t # ts)} \\
\text{note ith } = & \quad (t # ts) ! i = (p, is, xs, sb, D, O, R) \\
\text{show } \text{?case} & \\
\text{proof (cases i)} \\
\text{case 0 } & \\
\text{from ith 0 have } t: t = (p, is, xs, sb, D, O, R) \\
\text{by simp} \\
\text{show } \text{?thesis} & \\
\text{by (auto simp add: 0 t)} \\
\end{align*} \]

next

\[ \begin{align*}
\text{case (Suc n)} & \\
\text{obtain } p_i & \quad is_i \ O_i \ R_i \ D_i \ x_i \ sb_i \ where \ t: t = (p_i, is_i, xs_i, sb_i, D_i, O_i, R_i) \\
\text{by (cases t)} \\
\text{have n-bound: } n < \text{length ts} & \\
\text{using i-bound by (simp add: Suc)} \\
\text{have nth: } ts!n & \quad = (p, is, xs, sb, D, O, R) \\
\text{using ith by (simp add: Suc)} \\
\text{from Cons.hyps [OF n-bound nth]} & \\
\text{show } \text{?thesis} & \\
\text{by (auto simp add: Suc t)} \\
\text{qed} \\
\end{align*} \]

qed

lemma acquire-release-ownership-nth-update:

assumes R-subset: \( R \subseteq O \)

shows \( \forall i. \ [i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, R)] \)

\[ \implies \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot, O). \ O) \setminus \text{set} \ (ts[i:=(p',is',xs',sb',D',O \cup A \setminus R,R')])) \]

= \((\bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot, O). \ O) \setminus \text{set} \ ts)) \cup A \setminus R \)

proof (induct ts)
case Nil  
thus ?case  
by simp

next

case (Cons t ts)

note i-bound = i < length (t # ts)

note ith = (t # ts) ![ i = (p,is, xs, sb,D,O,R))

note dist = (ownership-distinct (t#ts))

then interpret ownership-distinct t#ts.

from dist

have dist' : ownership-distinct ts

by (rule ownership-distinct-tl)

show ?case

proof (cases i)

  case 0

  from ith 0 have t: t = (p,is,xs,sb,D,O,R)

  by simp

  have R ∩ (∪ ((λ(,-,-,-,-,O,). O) · set ts)) = {}

  proof

  -

  fix x

  assume x-R: x ∈ R

  assume x-ls: x ∈ (∪ ((λ(,-,-,-,-,O,). O) · set ts))

  then obtain j p_j is_j O_j R_j D_j xs_j sb_j where

  j-bound: j < length ts and

  jth: ts!j = (p_j,is_j,xs_j,sb_j,D_j,O_j,R_j) and

  x-in: x ∈ O_j

  by (fastforce simp add: in-set-conv-nth )

  from j-bound jth 0

  have (O ∪ all-acquired sb) ∩ (O_j ∪ all-acquired sb_j) = {}

  apply –

  apply (rule ownership-distinct [OF i-bound - ith, of Suc j])

  apply clarsimp+

  apply blast

  done

with x-R R-subset x-in have False

  by auto

  }

  thus ?thesis

by blast

qed

then

show ?thesis

  by (auto simp add: 0 t)

next

case (Suc n)

obtain p_n is_n O_n R_n D_n xs_n sb_n where t: t = (p_n,is_n,xs_n,sb_n,D_n,O_n,R_n)

  by (cases t)

have n-bound: n < length ts

using i-bound by (simp add: Suc)

189
have nth: ts!n = (p, is, xs, sb, D, O, R)
using ith by (simp add: Suc)

have R ∩ (O_i ∪ all-acquired sb_i) = {}
proof -
{
fix x
assume x-R: x ∈ R
assume x-owns_i: x ∈ (O_i ∪ all-acquired sb_i)
from t
have (O ∪ all-acquired sb) ∩ (O_i ∪ all-acquired sb_i) = {}
  apply -
  apply (rule ownership-distinct [OF i-bound - - ith, of 0])
  apply (auto simp add: Suc)
done
with x-owns_i x-R R-subset have False
  by auto
}
  thus ?thesis
by blast
qed

lemma (in valid-history) valid-history-nth-update:

[i < length ts; history-consistent θ (hd-prog p sb) sb ] ⇒
valid-history program-step (ts[i := (p, is, θ, sb, D, O, R)])
by (unfold-locales)
(auto dest: valid-history simp add: nth-list-update split: if-split-asm)

lemma (in valid-reads) valid-reads-nth-update:

[i < length ts; reads-consistent False O m sb ] ⇒
valid-reads m (ts[i := (p, is, xs, sb, D, O, R)])
by (unfold-locales)
(auto dest: valid-reads simp add: nth-list-update split: if-split-asm)

lemma (in load-tmps-distinct) load-tmps-distinct-nth-update:

[i < length ts; distinct-load-tmps is ] ⇒
load-tmps-distinct (ts[i := (p, is, xs, sb, D, O, R)])
by (unfold-locales)
(auto dest: load-tmps-distinct simp add: nth-list-update split: if-split-asm)

lemma (in read-tmps-distinct) read-tmps-distinct-nth-update:

[i < length ts; distinct-read-tmps sb ] ⇒
read-tmps-distinct (ts[i := (p, is, xs, sb, D, O, R)])

190
by (unfold-locales)
(auto dest: read-tmps-distinct simp add: nth-list-update split: if-split-asm)

lemma (in load-tmps-read-tmps-distinct) load-tmps-read-tmps-distinct-nth-update:
\[ [i < \text{length } ts; \text{load-tmps is } \cap \text{read-tmps sb} = \{\} \implies \text{load-tmps-read-tmps-distinct } (ts[i := (p,\text{is,xs,sb},D,O,R)])] \]
by (unfold-locales)
(auto dest: load-tmps-read-tmps-distinct simp add: nth-list-update split: if-split-asm)

lemma (in load-tmps-fresh) load-tmps-fresh-nth-update:
\[ [i < \text{length } ts; \text{load-tmps is } \cap \text{dom } \theta = \{\} \implies \text{load-tmps-fresh } (ts[i := (p,\text{is,}\theta,\text{sb},D,O,R)])] \]
by (fastforce dest: load-tmps-fresh simp add: nth-list-update split: if-split-asm)

fun flush-all-until-volatile-write::
\( (\text{'p,'p store-buffer,'dirty,'owns,'rels}) \text{ thread-config list } \Rightarrow \text{memory } \Rightarrow \text{memory} \)
where
flush-all-until-volatile-write [] m = m
| flush-all-until-volatile-write ((-, -, -, sb, -, -)#ts) m =
  flush-all-until-volatile-write ts (flush (takeWhile (Not ◦ \text{is-volatile-Write}_{sb}) sb) m)

fun share-all-until-volatile-write::
\( (\text{'p,'p store-buffer,'dirty,'owns,'rels}) \text{ thread-config list } \Rightarrow \text{shared } \Rightarrow \text{shared} \)
where
share-all-until-volatile-write [] S = S
| share-all-until-volatile-write ((-, -, -, sb, -, -)#ts) S =
  share-all-until-volatile-write ts (share (takeWhile (Not ◦ \text{is-volatile-Write}_{sb}) sb) S)

lemma takeWhile-dropWhile-real-prefix:
\[ [x \in \text{set } xs; \neg P x] \implies \exists y ys. \text{xs=takeWhile } P \text{ xs } @ y\#ys \land \neg P y \land \text{dropWhile } P \text{ xs } = y\#ys \]
by (induct xs) auto

lemma buffered-val-witness: buffered-val sb a = Some v \implies
\exists \text{volatile sop A L R W}. \text{Write}_{sb} \text{volatile a sop v A L R W} \in \text{set sb}
apply (induct sb)
apply simp
apply (clarsimp split: memref.splits option.splits if-split-asm)
apply blast
apply blast

191
lemma flush-append-Readsb:
\( \forall m. (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (sb @ [\text{Read}_{sb} \text{volatile} \ a \ t \ v])) \ m) = \text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ m \)
by (induct sb) (auto split: memref.splits)

lemma flush-append-write:
\( \forall m. (\text{flush} \ (sb @ [\text{Write}_{sb} \text{volatile} \ a \ sop \ v \ A \ L \ R \ W]) \ m) = (\text{flush} \ sb \ m) \ (a:=v) \)
by (induct sb) (auto split: memref.splits)

lemma flush-append-Progsb:
\( \forall m. (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (sb @ [\text{Prog}_{sb} p_1 \ p_2 \ mis])) \ m) = (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ m) \)
by (induct sb) (auto split: memref.splits)

lemma flush-append-Ghostsb:
\( \forall m. (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (sb @ [\text{Ghost}_{sb} A \ L \ R \ W])) \ m) = (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ m) \)
by (induct sb) (auto split: memref.splits)

lemma share-append:
\( \forall S. \text{share} \ (xs@ys) \ S = \text{share} \ ys \ (\text{share} \ xs \ S) \)
by (induct xs) (auto split: memref.splits)

lemma share-append-Readsb:
\( \forall S. (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (sb @ [\text{Read}_{sb} \text{volatile} \ a \ t \ v])) \ S) = (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ S) \)
by (induct sb) (auto split: memref.splits)

lemma share-append-Write_sb:
\( \forall S. (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (sb @ [\text{Write}_{sb} \text{volatile} \ a \ sop \ v \ A \ L \ R \ W])) \ S) = (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ S) \)
by (induct sb) (auto split: memref.splits)

lemma share-append-Progsb:
\( \forall S. (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (sb @ [\text{Prog}_{sb} p_1 \ p_2 \ mis])) \ S) = (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ S) \)
by (induct sb) (auto split: memref.splits)

lemma in-acquired-no-pending-write-outstanding-write:
\( a \in \text{acquired} \ False \ sb \ A \implies \text{outstanding-refs} \ \text{is-volatile-Write}_{sb} \ sb \neq \{\} \)
apply (induct sb)
apply (auto split: memref.splits)
done

lemma flush-buffered-val-conv:
\( \forall m. \text{flush} \ sb \ m \ a = (\text{case} \ \text{buffered-val} \ sb \ a \ \text{of} \ \text{None} \Rightarrow m \ a \ | \ \text{Some} \ v \Rightarrow v) \)
by (induct sb) (auto split: memref.splits option.splits)
lemma reads-consistent-unbuffered-snoc:
\[\forall m. \text{buffered-val } sb \ a = \text{None} \implies m \ a = v \implies \text{reads-consistent pending-write } O \ m \ sb \implies \]
\[\text{volatile } \implies \text{outstanding-refs is-volatile-Write}_{sb} \ sb = \{\} \implies \text{reads-consistent pending-write } O \ m \ (sb @ [\text{Read}_{sb} \ \text{volatile} \ a \ t \ v]) \]
by (simp add: reads-consistent-append flush-buffered-val-conv)

lemma reads-consistent-buffered-snoc:
\[\forall m. \text{buffered-val } sb \ a = \text{Some} \ v \implies \text{reads-consistent pending-write } O \ m \ sb \implies \]
\[\text{volatile } \implies \text{outstanding-refs is-volatile-Write}_{sb} \ sb = \{\} \implies \text{reads-consistent pending-write } O \ m \ (sb @ [\text{Read}_{sb} \ \text{volatile} \ a \ t \ v]) \]
by (simp add: reads-consistent-append flush-buffered-val-conv)

lemma reads-consistent-snoc-Write_{sb}:
\[\forall m. \text{reads-consistent pending-write } O \ m \ sb \implies \]
\[\text{reads-consistent pending-write } O \ m \ (sb @ [\text{Write}_{sb} \ \text{volatile} \ a \ sop \ v A L R W]) \]
by (simp add: reads-consistent-append)

lemma reads-consistent-snoc-Prog_{sb}:
\[\forall m. \text{reads-consistent pending-write } O \ m \ sb \implies \text{reads-consistent pending-write } O \ m \ (sb @ [\text{Prog}_{sb} \ p_1 \ p_2 \ mis]) \]
by (simp add: reads-consistent-append)

lemma reads-consistent-snoc-Ghost_{sb}:
\[\forall m. \text{reads-consistent pending-write } O \ m \ sb \implies \text{reads-consistent pending-write } O \ m \ (sb @ [\text{Ghost}_{sb} A L R W]) \]
by (simp add: reads-consistent-append)

lemma restrict-map-id [simp]:m |\� dom m = m
apply (rule ext)
subgoal for x
apply (case-tac m x)
apply (auto simp add: restrict-map-def domIff)
done
done

lemma flush-all-until-volatile-write-Read-commute:
shows \[\forall m i. [i < \text{length } ls \implies ls[i] = (p, \text{Read volatile } a \ t\#is, \vartheta_{sb}, D, O, R) \]
\[\implies \]
\[\text{flush-all-until-volatile-write} \]
\[\text{(ls[i] := (p, is, \vartheta_{t\rightarrow v}), sb @ [Read}_{sb} \ \text{volatile} \ a \ t \ v), D', O', R'}) \] m = \]
\text{flush-all-until-volatile-write } ls \ m
proof (induct ls)
case Nil thus ?case
by simp

next
case (Cons l ls)
note i-bound = \( i < \text{length} (l \# ls) \)
note ith = \((l \# ls)!i = (p, \text{Read volatile} a t \# is, \theta, sb, D, O, R)\)
show \(?\)case
proof (cases i)
case 0
from ith 0 have l: l = (p, \text{Read volatile} a t \# is, \theta, sb, D, O, R)
  by simp
thus \(?\)thesis
  by (simp add: 0 flush-append-Read sb del: fun-upd-apply)
next
case (Suc n)
obtain p l is l l O l l R l l D l l \theta l l sb l l where
  l: l = (p l l l is l l l l \theta l l l l sb l l l l D l l l l O l l l l R l l l l)
  by (cases l)
from i-bound ith
have flush-all-until-volatile-write\[(ls[\text{n := (p, is, } t \mapsto v), sb @ \text{Read volatile} a t v, D', O', R'])\]
  (flush (takeWhile (Not \circ \text{is-volatile-Write} sb l) \text{sb} l) m) =
  flush-all-until-volatile-write ls (flush (takeWhile (Not \circ \text{is-volatile-Write} sb l) \text{sb} l) m)
    apply -
    apply (rule Cons.hyps)
    apply (auto simp add: Suc l)
done

then
show \(?\)thesis
  by (simp add: Suc l del: fun-upd-apply)
qed
qed

lemma flush-all-until-volatile-write-append-Ghost-commute:
\[ \forall i m. \ [i < \text{length} ts; ts!i = (p, is, } \theta, sb, D, O, R)\]
  \implies \text{flush-all-until-volatile-write (ts[i := (p', is', } \theta', sb'\@[\text{Ghost sb A L R W}, D', O', R'])]}\]
m = flush-all-until-volatile-write ts m
proof (induct ts)
  case Nil thus \(?\)case
  by simp
next
case (Cons l ts)
note i-bound = \( i < \text{length} (l \# ts) \)
note ith = \((l \# ts)!i = (p, is, \theta, sb, D, O, R)\)
show \(?\)case
proof (cases i)
case 0
from ith 0 have l: l = (p, is, \theta, sb, D, O, R)
  by simp
thus \(?\)thesis
by (simp add: 0 flush-append-Ghost sb del: fun-upd-apply)

next
  case (Suc n)
  obtain p l O R D l \ where l = (p, l, O, R, D, sb)
  by (cases l)

  from i-bound ith
  have flush-all-until-volatile-write
    (ts[n := (p', l', o', \[Ghost A L R W\], D', O', R')])
    (flush (takeWhile (Not \circ is-volatile-Write sb) m) =
    flush-all-until-volatile-write ts
    (flush (takeWhile (Not \circ is-volatile-Write sb) m))
    apply -
    apply (rule Cons.hyps)
    apply (auto simp add: Suc l)
    done

  then show \thesis
    by (simp add: Suc l)
  qed

lemma update-commute:
assumes g-unchanged: \forall a m. a \notin G \longrightarrow g m a = m a
assumes g-independent: \forall a m. a \in G \longrightarrow g (f m) a = g m a
assumes f-unchanged: \forall a m. a \notin F \longrightarrow f m a = m a
assumes f-independent: \forall a m. a \in F \longrightarrow f (g m) a = f m a
assumes disj: G \cap F = {}
shows f (g m) = g (f m)
proof
  fix a
  show f (g m) a = g (f m) a
  proof (cases a \in G)
    case True
    with disj have a-notin-F: a \notin F
    by blast
    from f-unchanged [rule-format, OF a-notin-F, of g m]
    have f (g m) a = g m a .
    also
    from g-independent [rule-format, OF True]
    have \ldots = g (f m) a by simp
    finally show \thesis .
  next
    case False
    note a-notin-G = this
    show \thesis
    proof (cases a \in F)
      case True
      from f-independent [rule-format, OF True]
have \( f (g \ m) \ a = f \ m \ a \) by simp
also
from g-unchanged [rule-format, OF a-notin-G]
have \( \ldots = g (f \ m) \ a \)
by simp
finally show \(?thesis\).

next
case False
from f-unchanged [rule-format, OF False]
have \( f (g \ m) \ a = g \ m \ a \).
also
from g-unchanged [rule-format, OF a-notin-G]
have \( \ldots = m \ a \).
also
from f-unchanged [rule-format, OF False]
have \( \ldots = f \ m \ a \) by simp
also
from g-unchanged [rule-format, OF a-notin-G]
have \( \ldots = g (f \ m) \ a \)
by simp
finally show \(?thesis\).

qed

lemma update-commute':
assumes g-unchanged: \( \forall \ a \ m. \ a \notin G \rightarrow g \ m \ a = m \ a \)
assumes g-independent: \( \forall \ a \ m_1 \ m_2. \ a \in G \rightarrow g \ m_1 \ a = g \ m_2 \ a \)
assumes f-unchanged: \( \forall \ a \ m. \ a \notin F \rightarrow f \ m \ a = m \ a \)
assumes f-independent: \( \forall \ a \ m_1 \ m_2. \ a \in F \rightarrow f \ m_1 \ a = f \ m_2 \ a \)
assumes disj: \( G \cap F = \{\} \)
shows \( f (g \ m) = g (f \ m) \)
proof
from g-independent have g-ind': \( \forall \ a \ m. \ a \in G \rightarrow g (f \ m) \ a = g \ m \ a \) by blast
from f-independent have f-ind': \( \forall \ a \ m. \ a \in F \rightarrow f (g \ m) \ a = f \ m \ a \) by blast
from update-commute [OF g-unchanged g-ind' f-unchanged f-ind' disj]
show \(?thesis\).

qed

lemma flush-unchanged-addresses: \( \forall m. a \neq \text{outstanding-refs is-Write}_{sb} \ sb \rightarrow \text{flush sb m a} = m a \)
proof (induct sb)
case Nil thus \(?case\) by simp
next
case (Cons r sb)
not a-notin = \( \langle a \notin \text{outstanding-refs is-Write}_{sb} \ (r#sb) \rangle \)
show \(?case\)
proof (cases r)
case (Write_{sb} volatile a' sop v)
from a-notin obtain neq-a-a': a ≠ a' and a-notin': a ∉ outstanding-refs is-Write_{sb} sb
by (simp add: Write_{sb})
from Cons.hyps [OF a-notin', of m(a':=v)] neq-a-a'
show ?thesis
  apply (simp add: Write_{sb} del: fun-upd-apply)
  apply simp
  done

next
case (Read_{sb} volatile a' t v)
from a-notin obtain a-notin': a ∉ outstanding-refs is-Write_{sb} sb
  by (simp add: Read_{sb})
from Cons.hyps [OF a-notin', of m]
show ?thesis
  by (simp add: Read_{sb})

next
case Prog_{sb} with Cons show ?thesis by simp
next
case Ghost_{sb} with Cons show ?thesis by simp
qed
qed

lemma flushed-values-mem-independent:
\[ \forall m m'. a \in \text{outstanding-refs is-Write}_{sb} \Rightarrow \text{flush}_{sb} m' a = \text{flush}_{sb} m a \]
proof (induct sb)
case Nil thus ?case by simp

next
case (Cons r sb)
show ?case
proof (cases r)
case (Write_{sb} volatile a' sop' v')
have \text{flush}_{sb} (m'(a' := v')) a' = \text{flush}_{sb} (m(a' := v')) a'
proof (cases a' ∈ outstanding-refs is-Write_{sb} sb)
case True
  from Cons.hyps [OF this]
  show ?thesis.
next
case False
  from flush-unchanged-addresses [OF False]
  show ?thesis
by simp
qed
with Cons.hyps Cons.prems
show ?thesis
by (auto simp add: Write_{sb})

next
case Read_{sb} thus ?thesis using Cons
  by auto
next
case Prog_{sb} thus ?thesis using Cons
  by auto
lemma flush-all-until-volatile-write-unchanged-addresses:
  \(\forall m. a \notin \bigcup ((\lambda(-,-,-,-,-,-). \text{outstanding-refs is-Write}_{sb}
    \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})' \set \text{ls}) \implies \)
  flush-all-until-volatile-write \(ls m a = m a\)

proof (induct \(ls\))
  case Nil thus \(?\)case by simp

next
  case (Cons \(l\) \(ls\))
  obtain \(p\) is \(\mathcal{O} \mathcal{R} \mathcal{D}\) \(xs\) \(sb\) where \(l\) : \(l=(p,\text{xs},\text{sb},\mathcal{D},\mathcal{O},\mathcal{R})\)
    by (cases \(l\))
  note \(\langle a \notin \bigcup ((\lambda(-,-,-,-,-,-). \text{outstanding-refs is-Write}_{sb}
    \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})' \set (l#ls)\rangle\)
  then obtain \(a\)-notin-sb: \(a \notin \text{outstanding-refs is-Write}_{sb}\)
    (\(a\)-notin-ls: \(a \notin \bigcup ((\lambda(-,-,-,-,-,-). \text{outstanding-refs is-Write}_{sb}
      \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})' \set \text{ls})\)
    by (auto simp add: \(l\))

from Cons.hyps [OF \(a\)-notin-\(ls\)]
have flush-all-until-volatile-write \(ls\) (flush \(\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}\) \(m\)) \(a\)
  \(=\)
  (flush \(\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}\) \(m\)) \(a\).

also

from flush-unchanged-addresses [OF \(a\)-notin-sb]
have (flush \(\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}\) \(m\)) \(a\) = \(m\) \(a\).

finally
show \(?\)case
  by (simp add: \(l\))

qed

lemma notin-outstanding-non-volatile-takeWhile-lem:
  \(a \notin \text{outstanding-refs } (\text{Not } \circ \text{is-volatile}) \text{ sb}\)
  \(\implies a \notin \text{outstanding-refs is-Write}_{sb}\) (\(\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}\)

apply (induct \(sb\))
apply (auto simp add: is-Write\(_{sb}\)-def split: if-split asm memref.splits)
done

lemma notin-outstanding-non-volatile-takeWhile-lem':
a /∈ outstanding-refs is-non-volatile-Write sb

⇒

a /∈ outstanding-refs is-Write sb (takeWhile (Not o is-volatile-Write sb) sb)

apply (induct sb)
apply (auto simp add: is-Write sb-def split: if-split-asm memref.splits)
done

lemma notin-outstanding-non-volatile-takeWhile-Un-lem':
a /∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs (Not o is-volatile) sb) ' set ls)
⇒ a /∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs is-Write sb
(takeWhile (Not o is-volatile-Write sb) sb)) ' set ls)

proof (induct ls)
case Nil thus ?case by simp
next
case (Cons l ls)
obtain p is O R D xs sb where l = (p,is,xs,rb,D,O,R)
  by (cases l)

from Cons.prems
obtain
  a-notin-sb: a /∈ outstanding-refs (Not o is-volatile) sb and
  a-notin-ls: a /∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs (Not o is-volatile) sb) ' set ls)
  by (force simp add: o-apply)
from notin-outstanding-non-volatile-takeWhile-l lem [OF a-notin-sb]
Cons.hyps [OF a-notin-ls]
show ?case
  by (auto simp add: l simp del: o-apply)
qed

lemma flush-all-until-volatile-write-unchanged-addresses':
assumes notin: a /∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs (Not o is-volatile) sb) ' set ls)
sows flush-all-until-volatile-write ls m a = m a
using (auto intro: flush-all-until-volatile-write-unchanged-addresses)

proof (induct ls)
case Nil thus ?case by simp
next
case (Cons l ls)
obtain p is O R D xs sb where l = (p,is,xs,rb,D,O,R)
  by (cases l)

from Cons.prems
obtain
  a-in = a /∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs is-Write sb
(takeWhile (Not o is-volatile-Write sb) sb)) ' set ls)
⇒ flush-all-until-volatile-write ls m a = flush-all-until-volatile-write ls m a

proof (induct ls)
case Nil thus ?case by simp
next
case (Cons l ls)
obtain p is O R D xs sb where l = (p,is,xs,rb,D,O,R)
  by (cases l)

from Cons.prems
obtain
  a-in = a /∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs is-Write sb
(takeWhile (Not o is-volatile-Write sb) sb)) ' set ls)

proof (cases a ∈ ∪ ((λ(−,−,−,sb,−,−,−)). outstanding-refs is-Write sb
(takeWhile (Not o is-volatile-Write sb) sb)) ' set ls))
case True
from Cons.hyps [OF this]
show ?thesis
  by (simp add: l)
next
case False
with a-in
have a ∈ outstanding-refs is-Write_{ab} (takeWhile (Not ∘ is-volatile-Write_{ab}) sb)
  by (auto simp add: l)
from flushed-values-mem-independent [rule-format, OF this]
have flush (takeWhile (Not ∘ is-volatile-Write_{ab}) sb) m' a =
  flush (takeWhile (Not ∘ is-volatile-Write_{ab}) sb) m a.
with flush-all-until-volatile-write-unchanged-addresses [OF False]
show ?thesis
  by (auto simp add: l)
qed

lemma flush-all-until-volatile-write-buffered-val-conv:
assumes no-volatile-Write_{ab}: outstanding-refs is-volatile-Write_{ab} sb = {} shows
∀ i j. [ i < length ls; ls!i = (p, is, xs, sb, D, O, R);

  (∀ j. [ j < length ls. i ≠ j →
    (let (p, is, xs, sb, D, O, R) = ls!j
    in a ∉ outstanding-refs is-non-volatile-Write_{ab} (takeWhile (Not ∘ is-volatile-Write_{ab}) sb_j)) ] →
    flush-all-until-volatile-write ls m a =
    (case buffered-val sb a of None ⇒ m a | Some v ⇒ v)

proof (induct ls)
case Nil thus ?case
  by simp
next
case (Cons l ls)
  note i-bound = ⟨i < length (l#ls)⟩
  note ith = ⟨(l#ls)!i = (p, is, xs, sb, D, O, R)⟩
  note notin = ⟨∀ j. [ j < length (l#ls). i ≠ j →
    (let (p, is, xs, sb, D, O, R) = (l#ls)!j
    in a ∉ outstanding-refs is-non-volatile-Write_{ab} (takeWhile (Not ∘ is-volatile-Write_{ab}) sb_j)) ]→
    flush-all-until-volatile-write ls m a =
    (case buffered-val sb a of None ⇒ m a | Some v ⇒ v)
  ⟩
  show ?case
  proof (cases i)
  case 0
    from ith 0 have l: l = (p, is, xs, sb, D, O, R)
    by simp
    from no-volatile-Write_{ab} have take-all: takeWhile (Not ∘ is-volatile-Write_{ab}) sb = sb
    by (auto simp add: outstanding-refs-conv)
  have a ∉ (λ(⋅, ⋅, ⋅, sb, ⋅). outstanding-refs is-Write_{ab} (takeWhile (Not ∘ is-volatile-Write_{ab}) sb)) set ls) (is a ∉ ?LS)
proof
  assume a ∈ ?LS
  from in-Union-image-nth-conv [OF this]
  obtain j p j is j O j R j D j xs j sb j where
    j-bound: j < length ls and
    jth: ls j = (p j ,is j ,xs j ,sb j ,D j ,O j ,R j ) and
    a-in-j: a ∈ outstanding-refs is-Write sb (takeWhile (Not ◦ is-volatile-Write sb ) sb j)
  by fastforce
  from a-in-j obtain v' sop' A L R W where Write sb False a sop' v' A L R W ∈ set
    (takeWhile (Not ◦ is-volatile-Write sb ) sb j)
  apply (clarsimp simp add: outstanding-refs-conv )
  subgoal for x
  apply (case-tac x)
  apply clarsimp
  apply (frule set-takeWhileD)
  apply auto
  done
  done
  with notin [rule-format, of Suc j] j-bound jth
  show False
  by (force simp add: 0 outstanding-refs-conv is-non-volatile-Write sb -def
    split: memref.splits)
qed

proof
  have flush-all-until-volatile-write ls (flush (takeWhile (Not ◦ is-volatile-Write sb ) sb l) m) a = (flush sb m) a
    by (simp add: take-all)
  then
  show ?thesis
  by (simp add: 0 l take-all flush-buffered-val-conv)
next
  case (Suc n)
  obtain p l is l O l R l D l xs l sb l where l: l = (p l ,is l ,xs l ,sb l ,D l ,O l ,R l )
    by (cases l)
  from i-bound ith notin
  have flush-all-until-volatile-write ls
    (flush (takeWhile (Not ◦ is-volatile-Write sb ) sb l) m) a
      = (case buffered-val sb a of None ⇒
        (flush (takeWhile (Not ◦ is-volatile-Write sb ) sb l) m) a | Some v ⇒ v)
    apply –
    apply (rule Cons.hyps)
    apply (force simp add: Suc Let-def simp del: o-apply)+
  done

  moreover
  from notin [rule-format, of 0] l
  have a /∈ outstanding-refs is-non-volatile-Write sb (takeWhile (Not ◦ is-volatile-Write sb ) sb l)
    by (auto simp add: Let-def outstanding-refs-conv Suc )
  then

  201
have a \notin \text{outstanding-refs} \text{is-Write}_{sb} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \text{sb}_1) 
apply (\text{clarsimp simp add: outstanding-refs-conv is-Write}_{sb\_def} \text{split: memref.splits dest: set-takeWhileD})
apply (\text{frule set-takeWhileD})
apply force
done

from flush-unchanged-addresses [OF this]
have (flush (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \text{sb}_l) m) a = m a .

ultimately
show ?thesis
by (simp add: Suc l split: option.splits)
qed
qed

context program
begin

abbreviation sb-concurrent-step ::
\begin{align*}
(\text{'p}', p \text{ store-buffer}, \text{'dirty}, \text{'owns}, \text{'rels}, \text{'shared}) \text{ global-config} & \Rightarrow (\text{'p}', p \text{ store-buffer}, \text{'dirty}, \text{'owns}, \text{'rels}, \text{'shared}) \text{ global-config} \Rightarrow \text{bool} \\
(- \Rightarrow_{sb} - [60,60] 100)
\end{align*}
where
sb-concurrent-step \equiv
\begin{align*}
\text{computation.concurrent-step sb-memop-step store-buffer-step program-step (} & (\lambda p \ p' \text{ is sb. sb}) 
\end{align*}

term x \Rightarrow_{sb} Y

abbreviation (in program) sb-concurrent-steps::
\begin{align*}
(\text{'p}', p \text{ store-buffer}, \text{'dirty}, \text{'owns}, \text{'rels}, \text{'shared}) \text{ global-config} & \Rightarrow (\text{'p}', p \text{ store-buffer}, \text{'dirty}, \text{'owns}, \text{'rels}, \text{'shared}) \text{ global-config} \Rightarrow \text{bool} \\
(- \Rightarrow_{sb^*} - [60,60] 100)
\end{align*}
where
sb-concurrent-steps \equiv \text{sb-concurrent-step}^\ast


term x \Rightarrow_{sb^*} Y

abbreviation sbh-concurrent-step ::
\begin{align*}
(\text{'p}', p \text{ store-buffer}, \text{bool,owns,rels,shared}) \text{ global-config} & \Rightarrow (\text{'p}', p \text{ store-buffer}, \text{bool,owns,rels,shared}) \text{ global-config} \Rightarrow \text{bool} \\
(- \Rightarrow_{sbh} - [60,60] 100)
\end{align*}
where
sbh-concurrent-step \equiv
\begin{align*}
\text{computation.concurrent-step sbh-memop-step flush-step program-step (} & (\lambda p \ p' \text{ is sb. sb @ [Prog_{sb} p p' is] }) 
\end{align*}

term x \Rightarrow_{sbh} Y

202
abbreviation sbh-concurrent-steps::
  (p, p store-buffer, bool, owns, rels, shared) global-config \implies (p, p
store-buffer, bool, owns, rels, shared) global-config \implies bool
(- \Rightarrow_{sbh} - [60, 60] 100)
where
sbh-concurrent-steps \equiv sbh-concurrent-step\^**

term x \Rightarrow_{sbh}^* Y
end

lemma instrs-append-Read_{sb}:
iinstrs (sb@[Read_{sb} volatile a t v]) = instrs sb @ [Read volatile a t]
by (induct sb) (auto split: memref.splits)

lemma instrs-append-Write_{sb}:
iinstrs (sb@[Write_{sb} volatile a sop v A L R W]) = instrs sb @ [Write volatile a sop A L
R W]
by (induct sb) (auto split: memref.splits)

lemma instrs-append-Ghost_{sb}:
iinstrs (sb@[Ghost_{sb} A L R W]) = instrs sb @ [Ghost A L R W]
by (induct sb) (auto split: memref.splits)

lemma prog-instrs-append-Ghost_{sb}:
prog-instrs (sb@[Ghost_{sb} A L R W]) = prog-instrs sb
by (induct sb) (auto split: memref.splits)

lemma prog-instrs-append-Read_{sb}:
prog-instrs (sb@[Read_{sb} volatile a t v]) = prog-instrs sb
by (induct sb) (auto split: memref.splits)

lemma prog-instrs-append-Write_{sb}:
prog-instrs (sb@[Write_{sb} volatile a sop v A L R W]) = prog-instrs sb
by (induct sb) (auto split: memref.splits)

lemma hd-prog-append-Read_{sb}:
hd-prog p (sb@[Read_{sb} volatile a t v]) = hd-prog p sb
by (induct sb) (auto split: memref.splits)

lemma hd-prog-append-Write_{sb}:
hd-prog p (sb@[Write_{sb} volatile a sop v A L R W]) = hd-prog p sb
by (induct sb) (auto split: memref.splits)

lemma flush-update-other: \(\forall m. a \notin \text{outstanding-refs} (\text{Not } \circ \text{is-volatile})\ sb \implies
\text{outstanding-refs} (\text{is-volatile-Write}_{sb}) sb = \{\} \implies
\text{flush} sb (m(a:=v)) = (\text{flush} sb m)(a := v)
by (induct sb)
(auto split: memref.splits if-split-asm simp add: fun-upd-twist)

203
lemma flush-update-other': \( \forall m. a \notin \text{outstanding-refs} \,(\text{is-non-volatile-Write}_{sb}) \, sb \implies \text{outstanding-refs} \,(\text{is-volatile-Write}_{sb}) \, sb = \{\} \implies \text{flush} \,(m(a:=v)) = (\text{flush} \, sb) \,(a := v) \)

by (induct sb)
(auto split: memref.splits if-split-asm simp add: fun-upd-twist)

lemma flush-update-other'': \( \forall m. a \notin \text{outstanding-refs} \,(\text{is-non-volatile-Write}_{sb}) \, sb \implies \text{a} \notin \text{outstanding-refs} \,(\text{is-volatile-Write}_{sb}) \, sb \implies \text{flush} \,(m(a:=v)) = (\text{flush} \, sb) \,(a := v) \)

by (induct sb)
(auto split: memref.splits if-split-asm simp add: fun-upd-twist)

lemma flush-all-until-volatile-write-update-other:
\( \forall m. \forall j < \text{length} \, ts. \)
\( \text{(let } (-,\cdots,\text{sb}_j,\cdots,-) = \text{ts}!j \text{ in } \text{a} \notin \text{outstanding-refs} \,(\text{is-non-volatile-Write}_{sb}) \, \text{takeWhile} \,(\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{sb}_j) \)
\( \implies \text{flush-all-until-volatile-write} \, ts \,(m(a := v)) =
\,(\text{flush-all-until-volatile-write} \, ts) \,(a := v) \)

proof (induct ts)
case Nil thus \(?\)case
by simp
next
case (Cons t ts)
note notin = \( \forall j < \text{length} \,(t\#ts). \)
\( \text{(let } (-,\cdots,\text{sb}_j,\cdots,-) = \text{(t\#ts)}!j \text{ in } \text{a} \notin \text{outstanding-refs} \,(\text{is-non-volatile-Write}_{sb}) \, \text{takeWhile} \,(\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{sb}_j) \)
\( \implies \text{flush-all-until-volatile-write} \, \text{ts} \,(m(a := v)) =
\,(\text{flush-all-until-volatile-write} \, \text{ts}) \,(a := v) \)

obtain \( p_i \, \text{is}_1 \, \mathcal{O}_l \, \mathcal{R}_l \, D_l \, \text{xs}_i \, \text{sb}_i \) where \( t: \text{t} = (p_i,\text{is}_i,\text{xs}_i,\text{sb}_i,D_l,\mathcal{O}_l,\mathcal{R}_l) \)
by (cases t)

have no-write:
\( \text{outstanding-refs} \,(\text{is-volatile-Write}_{sb}) \, \text{takeWhile} \,(\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{sb}_i) = \{\} \)
by (auto simp add: outstanding-refs-conv dest: set-takeWhileD)

from notin [rule-format, of 0] t
have a-notin:
a \notin \text{outstanding-refs} \,(\text{is-non-volatile-Write}_{sb}) \, \text{takeWhile} \,(\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{sb}_i 
by (auto )

from flush-update-other' [OF a-notin no-write]
have \( \text{flush} \,(\text{takeWhile} \,(\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{sb}_i) \,(m(a := v))) =
\)
(flush (takeWhile (Not o is-volatile-Write \sb_\_sb) \sb_\_sb) \m)(a := v).

with Cons.hyps [OF notin', of (flush (takeWhile (Not o is-volatile-Write \sb_\_sb) \sb_\_sb) \m)]

show ?case
  by (simp add: t del: fun-upd-apply)

qed

lemma flush-all-until-volatile-write-append-non-volatile-write-commute:
  assumes no-volatile-Write \sb_\_sb: outstanding-refs is-volatile-Write \sb_\_sb \sb_\_sb = {}
  shows \( \forall m \ i. [i < \text{length ts}; ts!i = (p, is, xs, sb, D, O, R)]; \)
  \( \forall j < \text{length ts}. i \neq j \rightarrow \)
  \( \text{(let (\_,\_,\_,sb}_j,\_,\_,\_) = ts!j} \)
  \( \text{in a \notin outstanding-refs is-non-volatile-Write}_{\sb_\_sb} (\text{takeWhile (Not o is-volatile-Write}_{\sb_\_sb}) \sb_\_sb)\])
  \( \Rightarrow \) flush-all-until-volatile-write (ts[i := (p', is', xs, sb @ [Write _sb False a sop v A L R W]]) \m) =
  (flush-all-until-volatile-write ts \m)(a := v)

proof (induct ts)
  case Nil thus ?case
  next
  case (Cons t ts)
  note i-bound = \langle i < \text{length (t#ts)}\rangle
  note ith = \langle (t#ts)!i = (p, is, xs, sb, D, O, R)\rangle
  note notin = \langle \forall j < \text{length (t#ts)}. i \neq j \rightarrow \)
  \( \text{(let (\_,\_,\_,sb}_j,\_,\_,\_) = (t#ts)!j} \)
  \( \text{in a \notin outstanding-refs is-non-volatile-Write}_{\sb_\_sb} (\text{takeWhile (Not o is-volatile-Write}_{\sb_\_sb}) \sb_\_sb))\]
  show ?case
    proof (cases i)
      case 0
      from ith 0 have t: t = (p, is, xs, sb, D, O, R)
        by simp
      from no-volatile-Write \sb_\_sb have take-all: takeWhile (Not o is-volatile-Write \sb_\_sb) \sb_\_sb = sb
        by (auto simp add: outstanding-refs-conv)
      from no-volatile-Write \sb_\_sb have take-all': takeWhile (Not o is-volatile-Write \sb_\_sb) \sb_\_sb = sb @ [Write \sb_\_sb False a sop v A L W]
        by (auto simp add: outstanding-refs-conv)
      from notin have \forall j < \text{length ts}. \text{(let (\_,\_,\_,sb}_j,\_,\_,\_) = ts!j} \)
        \text{in a \notin outstanding-refs is-non-volatile-Write}_{\sb_\_sb} (\text{takeWhile (Not o is-volatile-Write}_{\sb_\_sb}) \sb_\_sb))
        by (auto simp add: 0)
      from flush-all-until-volatile-write-update-other [OF this]
      show ?thesis
        by (simp add: 0 t take-all' take-all flush-append-write del: fun-upd-apply)
next
  case (Suc n)
  obtain \( p_1 \, i_s \, O | R \, D \, \text{xs} \, \text{sb} \) where \( t : t = (p_1, i_s, \text{xs}, \text{sb}, D, O, R) \)
    by (cases t)
  from i-bound ith notin
  have \( \text{flush-all-until-volatile-write} \)
    \( (ts[n := (p', i_s', \text{xs}, \text{sb} \oplus [\text{Write}_{ab} \ False \ a \ L \ R \ W, D', O, R'])]) \)
    \( (\text{flush (takeWhile (Not} \circ \text{is-volatile-Write}_{ab} \text{sb}) \text{ sb}) \text{ m}) = \)
    \( (\text{flush-all-until-volatile-write ts} \)
      \( (\text{flush (takeWhile (Not} \circ \text{is-volatile-Write}_{ab} \text{sb}) \text{ sb}) \text{ m})) \)
      \( (a := v) \)
    apply --
    apply (rule Cons.hyps)
    apply (auto simp add: Suc simp del: o-apply)
  done

  then
  show \( ?\text{thesis} \)
    by (simp add: t Suc del: fun-upd-apply)

qed

lemma \( \text{flush-all-until-volatile-write-append-unflushed}: \)
\( \text{assumes} \, \text{volatile-Write}_{ab} : \neg \ \text{outstanding-refs} \, \text{is-volatile-Write}_{ab} \, \text{sb} = \{\} \)
\( \text{shows} \, \forall \, m \, i. \, [i < \text{length ts}; ts!i = (p, i_s, \text{xs}, \text{sb}, D, O, R)] \)
\( \implies \text{flush-all-until-volatile-write} \, (ts[i := (p', i_s', \text{xs}, \text{sb} \oplus \text{sbx}, D', O, R')]) \, m = \)
\( (\text{flush-all-until-volatile-write ts} \, m) \)
proof (induct ts)
  case Nil thus \( ?\text{case} \)
    by simp
next
  case (Cons l ts)
  note i-bound = \( \langle i < \text{length (l#ts)} \rangle \)
  note ith = \( \langle (l#ts)!i = (p, i_s, \text{xs}, \text{sb}, D, O, R) \rangle \)
  show \( ?\text{case} \)
  proof (cases i)
    case 0
    from ith 0 have \( l : l = (p, i_s, \text{xs}, \text{sb}, D, O, R) \)
      by simp
    from \( \text{volatile-Write}_{ab} \)
    obtain \( r \) where \( r \in \text{set sb} \, \text{and} \, \text{volatile-r: is-volatile-Write}_{ab} \, r \)
      by (auto simp add: outstanding-refs-conv)
    from \( \text{takeWhile-append1} \, [\text{OF} \, \text{r-in}, \, \text{of} \, (\text{Not} \circ \text{is-volatile-Write}_{ab}) \] \, \text{volatile-r} \)
    have \( (\text{flush (takeWhile (Not} \circ \text{is-volatile-Write}_{ab} \text{sb} \, \oplus \text{sbx}) \, m)) = \)
      \( (\text{flush (takeWhile (Not} \circ \text{is-volatile-Write}_{ab} \text{sb}) \, m)) \)
      by auto
    then
    show \( ?\text{thesis} \)
      by (simp add: 0 l)
next
case (Suc n)
  obtain p l is l R l D l xs l sb l where l: l = (p, is, D, O, R, l)
  by (cases l)

  from Cons.hyps [of n] i-bound ith
  show ?thesis
  by (simp add: l Suc)

qed

lemma flush-all-until-volatile-nth-update-unused:
shows \(\forall m. \left[ i < \text{length } ts; ts!i = (p, is, \theta, sb, D, O, R) \right] \Rightarrow \text{flush-all-until-volatile-write } (ts[i := (p', is', \theta', sb, D', O', R')]) m = \text{flush-all-until-volatile-write } ts m \)
proof (induct ts)
case Nil thus ?case
  by simp

next
case (Cons l ts)
  note i-bound = \(\langle i < \text{length } (l#ts) \rangle \)
  note ith = \(\langle (l#ts)!i = (p, is, \theta, sb, D, O, R) \rangle \)
  show ?case
    proof (cases i)
      case 0
      from ith 0 have l: l = (p, is, D, O, R)
        by simp
      show ?thesis
        by (simp add: 0 l)
    next
case (Suc n)
    obtain p l is l R l D l xs l sb l where l: l = (p, is, D, O, R, l)
      by (cases l)
    from Cons.hyps [of n] i-bound ith
    show ?thesis
      by (simp add: l Suc)
  qed

lemma flush-all-until-volatile-write-append-volatile-write-commute:
assumes no-volatile-Write sb: \{\}
shows \(\forall m. \left[ i < \text{length } ts; ts!i = (p, is, \theta, sb, D, O, R) \right] \Rightarrow \text{flush-all-until-volatile-write } (ts[i := (p', is', \theta', sb, D', O', R')]) m = \text{flush-all-until-volatile-write } ts m \)
proof (induct ts)
case Nil thus ?case
  by simp

next
case (Cons l ts)
note i-bound = \( i < \text{length (l#ts)} \)
note ith = \( (l#ts)!i = (p, is, \theta, sb, D, O, R) \)
show ?case
proof (cases i)
case 0
  from ith 0 have l: l = (p, is, \theta, sb, D, O, R)
  by simp
from no-volatile-Write_{sb}
have s1: takeWhile (Not \circ is-volatile-Write_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
from no-volatile-Write_{sb}
have s2: (takeWhile (Not \circ is-volatile-Write_{sb}) (sb @ [Write_{sb} True a sop v A L R W]))
  = sb
  by (auto simp add: outstanding-refs-conv)
show ?thesis
  by (simp add: 0 l s1 s2)
next
case (Suc n)
  obtain p l is l O l R l D l \theta l sb l where l: l = (p_l, is_l, \theta_l, sb_l, D_l, O_l, R_l)
    by (cases l)
  from Cons.hyps [of n] i-bound ith
  show ?thesis
    by (simp add: Suc)
qed
qed

lemma reads-consistent-update:
\[ \forall \text{pending-write } O \ m. \text{ reads-consistent pending-write } O \ m \ sb \iff \]
a \notin \text{outstanding-ref} (Not \circ \text{is-volatile}) sb \iff
  \text{reads-consistent pending-write } O \ (m(a := v)) \ sb
apply (induct sb)
apply simp
apply (clarsimp split: memrefplits if-split-asm
  simp add: fun-upd-twist)
subgoal for sb O m x1 addr val A R pending-write
apply (case-tac a=addr)
apply simp
apply (fastforce simp add: fun-upd-twist)
done
done

lemma (in program) history-consistent-hd-prog: \[ \forall p. \text{history-consistent } \theta \ p \ ' \ xs \iff \text{history-consistent } \theta \ (hd-prog p \ xs) \ xs \]
apply (induct xs)
apply simp
apply (auto split: memref.splits option.splits)
done

locale valid-program = program +
  fixes valid-prog
  assumes valid-prog-inv: \[ \theta \vdash p \rightarrow p'(p',is') \colon valid-prog \, p \rightarrow valid-prog \, p' \]

lemma (in valid-program) history-consistent-appendD:
\[
\forall \theta \, ys \, p. \ \forall \, sop \in \text{write-sops} \, xs, \ \text{valid-sop} \, sop \Rightarrow 
\begin{align*}
&\text{read-tmps} \, xs \cap \text{read-tmps} \, ys = \{\} \Rightarrow 
&\text{history-consistent} \, \theta \, p \, (xs@ys) \Rightarrow 
&(\text{history-consistent} \, (\theta |^\cdot \, (\text{dom} \, \theta \, \cap \, \text{read-tmps} \, ys)) \, p \, xs \land 
&\text{history-consistent} \, \theta \, (\text{last-prog} \, p \, xs) \, ys \land 
&\text{read-tmps} \, ys \cap \bigcup \text{write-sops} \, xs = \{\})
\end{align*}
\]

proof (induct xs)
  case Nil 
  proof (auto)
  next
  case (Cons x xs)
  note valid-sops = \( \forall \, sop \in \text{write-sops} \, (x \, \# \, xs), \ \text{valid-sop} \, sop \)\)
  note read-tmps-dist = \( \text{read-tmps} \, (x \# xs) \cap \text{read-tmps} \, ys = \{\} \)\)
  note consis = \( \text{history-consistent} \, \theta \, p \, ((x \# xs)@ys) \)\)
  show ?case
  proof (cases x)
  case (Write sb volatile a sop v)
  obtain D f \text{ where} \ sop: sop=(D,f)
  by (cases sop)
  from consis obtain
  D-tmps: D \subseteq \text{dom} \, \theta \, \land \, f-v: f \, \theta = v \land 

\begin{align*}
&\text{D-read-tmps} \, D \cap \text{read-tmps} \, (xs @ ys) = \{\} \land 
&\text{consis'}: \text{history-consistent} \, \theta \, p \, (xs @ ys) 
&\text{by (simp add: Write\_sb sop)}
\end{align*}

from valid-sops obtain
  valid-Df: \text{valid-sop} \, (D,f) \land 
  valid-sops': \forall \, sop \in \text{write-sops} \, xs, \, \text{valid-sop} \, sop 
  by (auto simp add: Write\_sb sop)
from valid-Df
  interpret \text{valid-sop} \, (D,f) \ .
from read-tmps-dist have read-tmps-dist': read-tmps \, xs \cap \text{read-tmps} \, ys = \{\} 
  by (simp add: Write\_sb)

from D-read-tmps have D-ys: D \cap \text{read-tmps} \, ys = \{\} 
  by (auto simp add: read-tmps-append)
with D-tmps have D-subset: D \subseteq \text{dom} \, \theta \, \cap \, \text{read-tmps} \, ys 
  by auto
moreover

from valid-sop |OF refl D-tmps|
  have f \, \theta = f \, (\theta |^\cdot \, D).
moreover
let \( ?\theta' = \emptyset |' (\text{dom } \emptyset - \text{read-tmps } ys) \)
from D-subset
have \( ?\theta' |' D = \emptyset |' D \)
apply \( - \)
apply (rule ext)
by (auto simp add: restrict-map-def)
moreover
from D-subset
have D-tmps': \( D \subseteq \text{dom } ?\theta' \)
by auto
ultimately
have f-v': \( f ?\theta' = v \)
using valid-sop \( \text{OF refl } D\text{-tmtps} \) f-v
by simp
from D-read-tmps
have D ∩ read-tmps xs = {}
by (auto simp add: read-tmps-append)
with Cons.hyps \( \text{OF valid-sops' read-tmps-dist' consis'} \) D-tmps D-subset f-v' D-ys
show ?thesis
by (auto simp add: Write sb sop)
next
case (Read sb volatile a t v)
from consis obtain
tmps-t: \( \emptyset t = \text{Some } v \) and
cosis': history-consistent \( \emptyset p (xs @ ys) \)
by (simp add: Read sb split: option.splits)
from read-tmps-dist
obtain t-ys: \( t \notin \text{read-tmps } ys \) and read-tmps-dist': \( \text{read-tmps } xs \cap \text{read-tmps } ys = {} \)
by (auto simp add: Read sb)
from valid-sops have valid-sops': \( \forall sop \in \text{write-sops } xs. \text{valid-sop } sop \)
by (auto simp add: Read sb)
from t-ys tmps-t
have \( (\emptyset |' (\text{dom } \emptyset - \text{read-tmps } ys)) t = \text{Some } v \)
by (auto simp add: restrict-map-def domIff)
with Cons.hyps \( \text{OF valid-sops' read-tmps-dist' consis'} \)
show ?thesis
by (auto simp add: Read sb)
next
case (Prog sb p1 p2 mis)
from consis obtain p1-p: \( p1 = p \) and
prog-step: \( \emptyset |' (\text{dom } \emptyset - \text{read-tmps } (xs @ ys)) |' p1 \rightarrow (p2, \text{mis}) \) and
cosis': history-consistent \( \emptyset p2 (xs @ ys) \)
by (auto simp add: Prog sb)
let \( ?\theta' = \emptyset |' (\text{dom } \emptyset - \text{read-tmps } ys) \)
have eq: \( ?\theta' |' (\text{dom } ?\theta' - \text{read-tmps } xs) = \emptyset |' (\text{dom } \emptyset - \text{read-tmps } (xs @ ys)) \)
apply (rule ext)

210
apply (auto simp add: read-tmps-append restrict-map-def domIff split: if-split-asm)
done

from valid-sops have valid-sops': ∀sop∈write-sops xs. valid-sop sop
  by (auto simp add: Progsb)
from read-tmps-dist
obtain read-tmps-dist': read-tmps xs ∩ read-tmps ys = {}
  by (auto simp add: Progsb)
from Cons.hyps [OF valid-sops' read-tmps-dist' consis'] p1-p prog-step eq
  show ?thesis
    by (simp add: Progsb)
next
case Ghostsb
  with Cons show ?thesis
    by auto
qed

lemma (in valid-program) history-consistent-appendI:
\[∀ \varnothing \ ys \ p. \ ∀ \text{sop} ∈ \text{write-sops} \ xs. \text{valid-sop} \ \text{sop} \implies\]
  history-consistent (\varnothing \ | \ (\text{dom} \ \varnothing \ − \ \text{read-tmps} \ ys)) p xs \implies
  history-consistent \varnothing \ (\text{last-prog} \ p \ xs) \ ys \implies
  read-tmps \ ys \ ∩ ∪ (\text{fst} \ ' \ \text{write-sops} \ xs) = {} \implies \text{valid-prog} \ p \implies
  history-consistent \varnothing \ p \ (\text{xs} @ \text{ys})\]
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
  note valid-sops = ⟨∀ sop∈write-sops (x # xs). valid-sop sop⟩
  note consis-xs = ⟨history-consistent (\varnothing \ | \ (\text{dom} \ \varnothing \ − \ \text{read-tmps} \ ys)) p (x # xs)⟩
  note consis-ys = ⟨history-consistent \varnothing \ (\text{last-prog} \ p (x # xs)) \ ys⟩
  note dist = ⟨read-tmps \ ys \ ∩ ∪ (\text{fst} \ ' \ \text{write-sops} (x # xs)) = {}⟩
  note valid-p = ⟨valid-prog p⟩
  show ?case
  proof (cases x)
case (Write sb volatile a sop v)
    obtain D f where sop={(D,f)}
      by (cases sop)
    from consis-xs obtain
      D-tmps: D ⊆ dom \varnothing − read-tmps ys and
      f-v: f (\varnothing \ | (\text{dom} \ \varnothing − read-tmps ys)) = v (is f ?\varnothing = v) and
      D-read-tmps: D ∩ read-tmps xs = {} and
      consis': history-consistent (\varnothing \ | (\text{dom} \ \varnothing − read-tmps ys)) p xs
    by (simp add: Writetb sop)
  from D-tmps D-read-tmps
  have D ∩ read-tmps (xs @ ys) = {}
    by (auto simp add: read-tmps-append)
  moreover
  from D-tmps have D-tmps': D ⊆ dom \varnothing

211
by auto

moreover
from valid-sops obtain
valid-Df: valid-sop (D,f) and
valid-sops': \forall sop\in write-sops xs. valid-sop sop
by (auto simp add: Write sb sop)
from valid-Df
interpret valid-sop (D,f).

from D-tmps
have tmps-eq: \emptyset |' ((\text{dom } \emptyset - \text{read-tmps ys}) \cap D) = \emptyset |' D
apply -
apply (rule ext)
apply (auto simp add: restrict-map-def)
done
from D-tmps
have f ?\emptyset = f (?\emptyset |' D)
apply -
apply (rule valid-sop [OF refl])
apply auto
done
with valid-sop [OF refl D-tmps'] f-v D-tmps

have f \emptyset = v
by (clarsimp simp add: tmps-eq)
moreover
from consis-ys have consis-ys': history-consistent \emptyset (last-prog p xs) ys
by (auto simp add: Write sb)
from dist have dist': read-tmps ys \cap \bigcup (fst ' write-sops xs) = {}
by (auto simp add: Write sb)

moreover note Cons.hyps [OF valid-sops' consis' consis-ys' dist' valid-p]

ultimately show ?thesis
by (simp add: Write sb sop)

next
case (Read sb volatile a t v)
from consis-xs obtain
t-v: (\emptyset |' (\text{dom } \emptyset - \text{read-tmps ys})) t = Some v and
consis-xs': history-consistent (\emptyset |' (\text{dom } \emptyset - \text{read-tmps ys})) p xs
by (clarsimp simp add: Read sb split: option.splits)
from t-v have \emptyset t = Some v
by (auto simp add: restrict-map-def split: if-split-asm)
moreover
from valid-sops obtain
valid-sops': \forall sop\in write-sops xs. valid-sop sop
by (auto simp add: Read sb)
from consis-ys have consis-ys': history-consistent \emptyset (last-prog p xs) ys
by (auto simp add: Read sb)
from dist have dist': read-tmps ys ∩ \( \bigcup \text{fst' write-sops xs} \) = \{
by (auto simp add: Read_{sb})

note Cons.hyps [OF valid-sops' consis-xs' consis-ys' dist' valid-p]
ultimately
show ?thesis
by (simp add: Read_{sb})

next
case (Prog_{sb} p_1 p_2 mis)
let ?\theta = \\theta |' (dom \\theta - read-tmps ys)
from consis-xs obtain
p_1-p: p_1 = p and
prog-step: ?\theta |' (dom ?\theta - read-tmps xs) |- p_1 \rightarrow_p (p_2, mis) and
consis': history-consistent ?\theta p_2 xs
by (auto simp add: Prog_{sb})

have eq: ?\theta |' (dom ?\theta - read-tmps xs) = ?\theta |' (dom ?\theta - read-tmps (xs @ ys))
apply (rule ext)
apply (auto simp add: read-tmps-append restrict-map-def domIff split: if-split-asm)
done

from prog-step eq
have \\theta |' (dom \\theta - read-tmps (xs @ ys)) = \\theta |' (dom \\theta - read-tmps (xs @ ys)) = \\Rightarrow valid-prog p = \Rightarrow history-consistent \\theta p (xs@ys)
by simp
moreover
from valid-sops obtain
valid-sops': \forall sop \in write-sops xs. valid-sop sop
by (auto simp add: Prog_{sb})
from consis-ys have consis-ys': history-consistent \\theta (last-prog p_2 xs) ys
by (auto simp add: Prog_{sb})
from dist have dist': read-tmps ys ∩ \( \bigcup \text{fst' write-sops xs} \) = \{
by (auto simp add: Prog_{sb})

note Cons.hyps [OF valid-sops' consis' consis-ys' dist' valid-prog-inv [OF prog-step valid-p [simplified p_1-p [symmetric]]]]
ultimately
show ?thesis
by (simp add: Prog_{sb} p_1-p)

next
case Ghost_{sb}
with Cons show ?thesis
by auto
qed

qed

lemma (in valid-program) history-consistent-append-conv:
\forall \theta ys p. \forall sop \in write-sops xs. valid-sop sop \Rightarrow
read-tmps xs ∩ read-tmps ys = \{
\Rightarrow valid-prog p \Rightarrow
history-consistent \\theta p (xs@ys) =
(history-consistent \ (\theta)' |' (dom \\theta - read-tmps ys)) p xs ∧
apply rule
apply (rule history-consistent-appendD, assumption+)
apply (rule history-consistent-appendI)
apply auto
done

lemma instrs-takeWhile-dropWhile-conv:
  instrs xs = instrs (takeWhile P xs) @ instrs (dropWhile P xs)
by (induct xs) (auto split: memref.splits)

lemma (in program) history-consistent-hd-prog-p:
  \( \forall p. \text{history-consistent} \theta p \Rightarrow p = \text{hd-prog} p \times \)
by (induct xs) (auto split: memref.splits option.splits)

lemma instrs-append: \( \forall y. \text{instrs} (xs@ys) = \text{instrs} xs @ \text{instrs} ys \)
by (induct xs) (auto split: memref.splits)

lemma prog-instrs-append: \( \forall y. \text{prog-instrs} (xs@ys) = \text{prog-instrs} xs @ \text{prog-instrs} ys \)
by (induct xs) (auto split: memref.splits)

lemma prog-instrs-empty: \( \forall r \in \text{set} xs. \neg \text{is-Prog} sb r \Rightarrow \text{prog-instrs} xs = [\] \)
by (induct xs) (auto split: memref.splits)

lemma length-dropWhile [termination-simp]: length (dropWhile P xs) \( \leq \) length xs
by (induct xs) auto

lemma prog-instrs-filter-is-Prog: \( \forall [\text{filter} (\text{is-Prog} sb) xs) = \text{prog-instrs} xs \)
by (induct xs) (auto split: memref.splits)

lemma Cons-to-snoc: \( \forall x. \exists y. (x\#xs) = (ys@[y]) \)
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x1 xs)
from Cons [of x1] obtain ys y where x1#xs = ys @ [y]
by auto
then
show ?case
by simp
qed

lemma causal-program-history-Read:
assumes causal-Read: causal-program-history (Read volatile a t \# is_{sb}) sb
shows causal-program-history is_{sb} (sb @ [Read_{sb} volatile a t v])
proof
fix \( sb_1 \) \( sb_2 \)
assume sb: sb \( @ [\text{Read}_{ab} \text{ volatile a t v}] = sb_1 @ sb_2 \)
from causal-Read
interpret causal-program-history Read volatile a t \# is_{sb} sb .
show \( \exists \text{is. instrs} \) sb_2 @ is_{sb} = is @ prog-instrs sb_2
proof (cases sb_2)
  case Nil
  thus \(?\)thesis
  by simp
next
case (Cons r sb' )
from Cons-to-snoc [of r sb'] Cons obtain ys y where sb_2-snoc: sb_2=ys\@[y]
  by auto
with sb obtain y: y = Read_{sb} volatile a t v and sb: sb = sb_1@ys
  by simp

from causal-program-history [OF sb] obtain is where
  instrs ys \@ Read volatile a t \# is_{sb} = is \@ prog-instrs ys
  by auto
then show \(?\)thesis
  by (simp add: sb_2-snoc y instrs-append prog-instrs-append)
qed

lemma causal-program-history-Write:
assumes causal-Write: causal-program-history (Write volatile a sop A L R W\# is_{sb}) sb
shows causal-program-history is_{sb} (sb \@ [Write_{sb} volatile a sop v A L R W])
proof
fix \( sb_1 \) \( sb_2 \)
assume sb: sb \@ [Write_{sb} volatile a sop v A L R W] = sb_1 @ sb_2
from causal-Write
interpret causal-program-history Write volatile a sop A L R W\# is_{sb} sb .
show \( \exists \text{is. instrs} \) sb_2 @ is_{sb} = is @ prog-instrs sb_2
proof (cases sb_2)
  case Nil
  thus \(?\)thesis
  by simp
next
case (Cons r sb' )
from Cons-to-snoc [of r sb'] Cons obtain ys y where sb_2-snoc: sb_2=ys\@[y]
  by auto
with sb obtain y: y = Write_{sb} volatile a sop v A L R W and sb: sb = sb_1@ys
  by simp

from causal-program-history [OF sb] obtain is where
  instrs ys \@ Write volatile a sop A L R W\# is_{sb} = is \@ prog-instrs ys
  by auto
then show \(?\)thesis
  by (simp add: sb_2-snoc y instrs-append prog-instrs-append)
Lemma causal-program-history-ProgM:

assumes causal-Write: causal-program-history isM sb

shows causal-program-history (isM @mis) (sb @ [ProgM p1 p2 mis])

proof
  fix sb1 sb2
  assume sb: sb @ [ProgM p1 p2 mis] = sb1 @ sb2
  from causal-Write
  interpret causal-program-history isM sb .
  show \exists is. instrs sb2 @ (isM @mis) = is @ prog-instrs sb2
  proof (cases sb2)
    case Nil
    thus \?thesis
    by simp
  next
    case (Cons r sb')
    from Cons-to-snoc [of r sb'] Cons obtain ys y where sb2-snoc: sb2=ys@[y]
    by auto
    with sb obtain y: y = ProgM p1 p2 mis and sb: sb = sb1@ys
    by simp
    from causal-program-history [OF sb]
    obtain is where
      instrs ys @ (isM @mis) = is @ prog-instrs (ys@[ProgM p1 p2 mis])
    by (auto simp add: prog-instrs-append)
    then show \?thesis
    by (simp add: sb2-snoc y instrs-append prog-instrs-append)
  qed

Lemma causal-program-history-Ghost:

assumes causal-GhostM: causal-program-history (Ghost A L R W # isM) sb

shows causal-program-history isM (sb @ [GhostM A L R W])

proof
  fix sb1 sb2
  assume sb: sb @ [GhostM A L R W] = sb1 @ sb2
  from causal-GhostM
  interpret causal-program-history Ghost A L R W # isM sb .
  show \exists is. instrs sb2 @ isM = is @ prog-instrs sb2
  proof (cases sb2)
    case Nil
    thus \?thesis
    by simp
  next
    case (Cons r sb')
    from Cons-to-snoc [of r sb'] Cons obtain ys y where sb2-snoc: sb2=ys@[y]
    by auto
    with sb obtain y: y = GhostM A L R W and sb: sb = sb1@ys
    by simp
from causal-program-history [OF sb] obtain is where 
iinstrs ys @ Ghost A L R W # is sb = is @ prog-instrs ys
by auto
then show ?thesis
  by (simp add: sb2-snoc y instrs-append prog-instrs-append)
qed
qed

lemma hd-prog-last-prog-end: [p = hd-prog p sb ; last-prog p sb = p sb] ⇒ p = hd-prog p sb
  by (induct sb) (auto split: memref.splits)
lemma hd-prog-idem: hd-prog (hd-prog p xs) xs = hd-prog p xs
  by (induct xs) (auto split: memref.splits)
lemma last-prog-idem: last-prog (last-prog p sb) sb = last-prog p sb
  by (induct sb) (auto split: memref.splits)
lemma last-prog-hd-prog-append: last-prog (hd-prog p sb (sb@sb′)) sb = last-prog (hd-prog p sb sb′) sb
  apply (induct sb)
  apply (auto split: memref.splits)
done
lemma last-prog-hd-prog: last-prog (hd-prog p xs) xs = last-prog p xs
  by (induct xs) (auto split: memref.splits)

lemma last-prog-append-Read sb:
  ∃p. last-prog p (sb @ [Read sb volatile a t v]) = last-prog p sb
  by (induct sb) (auto split: memref.splits)

lemma last-prog-append-Write sb:
  ∃p. last-prog p (sb @ [Write sb volatile a sop v A L R W]) = last-prog p sb
  by (induct sb) (auto split: memref.splits)

lemma last-prog-append-Prog sb:
  ∃x. last-prog x (sb @ [Prog sb p p′ mis]) = p′
  by (induct sb) (auto split: memref.splits)

lemma hd-prog-append-Prog sb:
  hd-prog x (sb @ [Prog sb p p′ mis]) = hd-prog p sb
  by (induct sb) (auto split: memref.splits)

lemma hd-prog-last-prog-append-Prog sb:
  ∃p′. hd-prog p xs = p′ ⇒ last-prog p′ xs = p1 ⇒

217
hd-prog p' (xs @ [Prog_{sb} p_1 p_2 mis]) = p'
apply (induct xs)
apply (auto split: memref.splits)
done

lemma hd-prog-append-Ghost_{sb}:
  hd-prog p (sb@[Ghost_{sb} A R L W]) = hd-prog p sb
by (induct sb) (auto split: memref.splits)

lemma last-prog-append-Ghost_{sb}:
  ∀p. last-prog p (sb@[Ghost_{sb} A L R W]) = last-prog p sb
by (induct sb) (auto split: memref.splits)

lemma dropWhile-all-False-conv:
  ∀x ∈ set xs. ¬P x ⇒ dropWhile P xs = xs
by (induct xs) auto

lemma dropWhile-append-all-False:
  ∀y ∈ set ys. ¬P y ⇒ dropWhile P (xs@ys) = dropWhile P xs @ ys
apply (induct xs)
apply (auto simp add: dropWhile-all-False-conv)
done

lemma reads-consistent-append-first:
  ∀m ys. reads-consistent pending-write O m (xs @ ys) ⇒ reads-consistent pending-write O m xs
by (clarsimp simp add: reads-consistent-append)

lemma reads-consistent-takeWhile:
assumes consis: reads-consistent pending-write O m sb
shows reads-consistent pending-write O m (takeWhile P sb)
using reads-consistent-append [where xs=(takeWhile P sb) and ys=(dropWhile P sb)]
consis
apply (simp add: reads-consistent-append)
done

lemma flush-flush-all-until-volatile-write-Write_{sb}-volatile-commute:
  ∀i m. [i < length ts; ts!i=(p,is,xs,Write_{sb} True a sop v A L R W#sb,D,O,R);]
  ∀i < length ts. (∀ j < length ts. i ≠ j →
    (let (⋅,⋅,⋅,sb_{j,⋅,⋅,⋅}) = ts!i;
    (⋅,⋅,⋅,sb_{j,⋅,⋅,⋅}) = ts!j
     in outstanding-refs is-Write_{sb} sb_{j} ∩
     outstanding-refs is-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_{j}) =
     {()});
  ∀j < length ts. i ≠ j →
    (let (⋅,⋅,⋅,sb_{j,⋅,⋅,⋅}) = ts!j in a ∉ outstanding-refs is-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_{j}))]
  ⇒
flush (takeWhile (Not ◦ is-volatile-Write sb) sb)  
((flush-all-until-volatile-write ts m)(a := v)) =  
flush-all-until-volatile-write (ts[i := (p,is,xs, sb, D', O', R')])  
(m(a := v))

proof (induct ts)
case Nil thus ?case  
  by simp
next
  case (Cons l ts)
  note i-bound = ⟨i < length (l#ts)⟩
  note ith = ⟨(l#ts)!i = (p,is,xs, Write sb True a sop v A L R W#sb, D, O, R)⟩
  note disj = ⟨∀ i < length (l#ts). (∀ j < length (l#ts). i ≠ j →  
    (let (_,_,_,sb,j,_,_,_) = (l#ts)!j  
      in outstanding-refs is-Write sb j ∩  
      outstanding-refs is-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb j) =  
    {⟨⟩}))⟩
  note a-notin = ⟨∀ j < length (l#ts). i ≠ j →  
    (let (_,_,_,sb,j,_,_,_) = (l#ts)!j
      in a /∈ outstanding-refs is-Write sb j)⟩
  show ?case
  proof (cases i)
  case 0
  from ith 0 have l = (p,is,xs,Write sb True a sop v A L R W#sb, D, O, R)
    by simp
  have a-notin-ts: a /∈ ∪(λ(_,_,_,sb,j,_,_,_). outstanding-refs is-Write sb  
    (takeWhile (Not ◦ is-volatile-Write sb) sb j)) ' set ts)
    (is a /∈ ?U)
    proof
      assume a ∈ ?U
      from in-Union-image-nth-conv [OF this]
      obtain j p j is j O j R j D j xs j sb j where
      j-bound: j < length ts and
      jth: ts![j] = (p,j,is,j,xs,j, D,j,O,j,R,j) and
      a-in-j: a ∈ outstanding-refs is-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb j)
    by fastforce
    from a-notin [rule-format, of Suc j] j-bound 0 a-in-j
    show False
    by (auto simp add: jth)
    qed
  from a-notin-ts
  have (flush-all-until-volatile-write ts m)(a := v) =  
    flush-all-until-volatile-write ts (m(a := v))
    apply 
    apply (rule update-commute' [where F=\{a\} and G=?U and  
    g=flush-all-until-volatile-write ts])
    apply (auto intro: flush-all-until-volatile-wirte-mem-independent
      flush-all-until-volatile-write-unchanged-addresses)
    done
moreover

let \( ?SB = \text{outstanding-refs is-Write}_{sb} \) (\text{takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb})

have U-SB-disj: \( ?U \cap ?SB = \emptyset \)
proof
{
fix \( a' \)
assume a'-in-U: \( a' \in \ ?U \)
have \( a' \notin ?SB \)
proof
assume a'-in-SB: \( a' \in \ ?SB \)
hence a'-in-SB': \( a' \in \text{outstanding-refs is-Write}_{sb} \)
apply (clarsimp simp add: outstanding-refs-conv)
apply (drule set-takeWhileD)
subgoal for \( x \)
apply (rule-tac x=x in exI)
apply (auto simp add: is-Write_{sb}-def split:memref.splits)
done
done
from in-Union-image-nth-conv [OF a'-in-U]
obtain j p_j \( j \in \text{is}_{j} \) \( R_{j} \) \( D_{j} \) \( x_{j} \) sb_j where
j-bound: \( j < \text{length ts} \) and
jth: \( \text{ts}[j] = (p_j, is_j, x_j, sb_j, D_j, C_j, R_j) \) and
a'-in-j: \( a' \in \text{outstanding-refs is-Write}_{sb} \) (\text{takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb}_j)
by fastforce
from disj [rule-format, of 0 Suc j] 0 j-bound a'-in-SB' a'-in-j jth l
show False
by auto
qed

moreover

fix \( a' \)
assume a'-in-SB: \( a' \in \ ?SB \)
hence a'-in-SB': \( a' \in \text{outstanding-refs is-Write}_{sb} \)
apply (clarsimp simp add: outstanding-refs-conv)
apply (drule set-takeWhileD)
subgoal for \( x \)
apply (rule-tac x=x in exI)
apply (auto simp add: is-Write_{sb}-def split:memref.splits)
done
done
have \( a' \notin ?U \)
proof
assume a' \in \ ?U
from in-Union-image-nth-conv [OF this]
obtain j p_j \( j \in \text{is}_{j} \) \( R_{j} \) \( D_{j} \) \( x_{j} \) sb_j where
j-bound: \( j < \text{length } ts \) and
jth: \( ts!j = (p_j, is_j, xs_j, sb_j, D_j, R_j, O_j) \) and
a'-'in-j: \( a' \in \text{outstanding-refs is-Write}_{sb} \) (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb\(_j\)) by fastforce

from disj [rule-format, of 0 Suc j] j-bound a'-'in-SB' a'-'in-j jth 1 show False by auto qed

ultimately show ?thesis by blast qed

have flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb)
  (flush-all-until-volatile-write ts (m(a := v))) =
  flush-all-until-volatile-write ts
  (flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb) (m(a := v)))
apply (rule update-commute'[where g = flush-all-until-volatile-write ts ,
  OF - - - - U-SB-disj])
apply (auto intro: flush-all-until-volatile-wirte-mem-independent
  flush-all-until-volatile-write-unchanged-addresses
  flush-unchanged-addresses
  flushed-values-mem-independent simp del: o-apply)
done

ultimately
have flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb)
  ((flush-all-until-volatile-write m)(a := v)) =
  flush-all-until-volatile-write ts
  (flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb) (m(a := v)))
by simp

then show ?thesis
  by (auto simp add: l 0 o-def simp del: fun-upd-apply)
next
  case (Suc n)

obtain p_i \ is_i \ R_i \ D_j \ x_s \ sb_l where l: l = (p_i, is_i, x_s, sb_l, D_j, O_l, R_l)
  by (cases l)

from i-bound ith disj a-notin
have flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb)
  ((flush-all-until-volatile-write ts
    (flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb_l) m))
  (a := v)) =
  flush-all-until-volatile-write (ts[n := (p, is, x_s, sb, D', O', R')])
  ((flush (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb} \)) sb_l) m)(a := v))
apply –
apply (rule Cons.hyps)
apply (force simp add: Suc Let-def simp del: o-apply)+
done

moreover

let ?SB = outstanding-refs is-Write\_sb (takeWhile (Not \circ is-volatile-Write\_sb) \_sb)

have a \notin \?SB

proof

assume a \in \?SB

with a-notin [rule-format, of 0]

show False

by (auto simp add: l Suc)

qed

then

have ((flush (takeWhile (Not \circ is-volatile-Write\_sb) \_sb) \_m)(a := v)) =

(flush (takeWhile (Not \circ is-volatile-Write\_sb) \_sb) (m(a := v)))

apply −

apply (rule update-commute'[where \_m=m and F={a} and G=?SB])

apply (auto intro:

flush-unchanged-addresses

flushed-values-mem-independent simp del: o-apply)

done

ultimately

show \?thesis

by (simp add: l Suc del: fun-upd-apply o-apply)

qed

qed

lemma (in program)

\(\forall \_sb' \_p. \text{history-consistent } \vartheta \ (\text{hd-prog } \_p (\_sb @ \_sb')) (\_sb @ \_sb') \Rightarrow\)

last-prog (\_sb @ \_sb') = \_p \Rightarrow\)

last-prog (\text{hd-prog } \_p (\_sb @ \_sb')) \_sb = \text{hd-prog } \_p \_sb'

proof (induct \_sb)

case Nil thus \?case by simp

next

case (Cons \_r \_sb\_1)

have consis: history-consistent \vartheta \ (\text{hd-prog } \_p ((\_r \# \_sb\_1) @ \_sb')) ((\_r \# \_sb\_1) @ \_sb')

by fact

have last-prog: last-prog (\_p ((\_r \# \_sb\_1) @ \_sb')) = \_p by fact

show ?case

proof (cases \_r)
case \texttt{Write}_{sb} \ with \ \texttt{Cons} show \ \texttt{thesis} by \ auto

next

case \texttt{Read}_{sb} \ with \ \texttt{Cons} show \ \texttt{thesis} by \ (auto \ split: \ \texttt{option.splits})

next

\begin{itemize}
  \item \texttt{case} \ \texttt{(Prog}_{sb} \ p_{1} \ p_{2} \ \texttt{is)}
  \item \texttt{from} \ \texttt{last-prog} \ \texttt{have} \ \texttt{last-prog-p}_{2}: \ \texttt{last-prog} \ p_{2} \ (sb_{1} \ @ \ sb^{'}) = p \\
  \quad \texttt{by} \ (\texttt{simp add: \texttt{Prog}_{sb}})
  \item \texttt{from} \ \texttt{consis} \ \texttt{obtain} \ \texttt{consis'}: \ \texttt{history-consistent} \ \emptyset \ p_{2} \ (sb_{1} \ @ \ sb^{'}) \\
  \quad \texttt{by} \ (\texttt{simp add: \texttt{Prog}_{sb}})
\end{itemize}

\hspace{1em} \texttt{hence} \ \texttt{history-consistent} \ \emptyset \ (\texttt{hd-prog} \ p_{2} \ (sb_{1} \ @ \ sb^{'})) \ (sb_{1} \ @ \ sb^{'}) \\
\hspace{1em} \texttt{by} \ (\texttt{rule \ \texttt{history-consistent-hd-prog}})

\item \texttt{from} \ \texttt{Cons.hyps} \ [\texttt{OF \ this \ ]}
\item \texttt{have} \ \texttt{last-prog} \ p_{2} \ sb_{1} = \texttt{hd-prog} \ p \ sb'
\item \texttt{oops}

\texttt{lemma} \ \texttt{last-prog-to-last-prog-same}: \ \exists \ p'. \ \texttt{last-prog} \ p' \ sb = p \Longrightarrow \ \texttt{last-prog} \ p \ sb = p \\
\texttt{by} \ (\texttt{induct \ sb}) \ (\texttt{auto \ split: \ \texttt{memref.splits}})

\texttt{lemma} \ \texttt{last-prog-hd-prog-same}: \ [\texttt{last-prog} \ p' \ sb = p; \ \texttt{hd-prog} \ p' \ sb = p'] 
\Longrightarrow \ \texttt{hd-prog} \ p \ sb = p' \\
\texttt{by} \ (\texttt{induct \ sb}) \ (\texttt{auto \ split: \ \texttt{memref.splits}})

\texttt{lemma} \ \texttt{(in \ program) last-prog-hd-prog-append}': \ \exists \sb'. \ \texttt{history-consistent} \ \emptyset \ (\texttt{hd-prog} \ p \ (sb@sb')) \ (sb@sb') \Longrightarrow \\
\texttt{last-prog} \ p \ (sb@sb') = p \Longrightarrow \\
\texttt{last-prog} \ (\texttt{hd-prog} \ p \ sb') \ sb = \texttt{hd-prog} \ p \ sb'

\texttt{proof} \ (\texttt{induct \ sb})
\item \texttt{case} \ Nil \ \texttt{thus} \ ?\texttt{case} \ \texttt{by simp}
\item \texttt{next}
\item \texttt{case} \ \texttt{(Cons \ r \ sb_{1})}
\item \texttt{have} \ \texttt{consis}: \ \texttt{history-consistent} \ \emptyset \ (\texttt{hd-prog} \ p \ ((r \ # \ \texttt{sb}_{1}) \ @ \ sb^{'})) \ ((r \ # \ \texttt{sb}_{1}) \ @ \ sb^{'}) \\
\quad \texttt{by} \ \texttt{fact}
\item \texttt{have} \ \texttt{last-prog}: \ \texttt{last-prog} \ p \ ((r \ # \ \texttt{sb}_{1}) \ @ \ sb^{'}) = p \ \texttt{by fact}
\item \texttt{show} \ ?\texttt{case}
\item \texttt{proof} \ (\texttt{cases \ r})
  \item \texttt{case} \ \texttt{Write}_{sb} \ \texttt{with} \ \texttt{Cons} show \ \texttt{thesis} \ \texttt{by} \ \texttt{auto}
\item \texttt{next}
\item \texttt{case} \ \texttt{Read}_{sb} \ \texttt{with} \ \texttt{Cons} show \ \texttt{thesis} \ \texttt{by} \ (\texttt{auto \ split: \ \texttt{option.splits}})
\item \texttt{next}
\item \texttt{case} \ \texttt{(Prog}_{sb} \ p_{1} \ p_{2} \ \texttt{is)}
\item \texttt{from} \ \texttt{last-prog} \ \texttt{have} \ \texttt{last-prog-p}_{2}: \ \texttt{last-prog} \ p_{2} \ (sb_{1} \ @ \ sb^{'}) = p

\texttt{done}
by (simp add: Prog sb)
from last-prog-to-last-prog-same [OF this]
have last-prog-p: last-prog p (sb_1 @ sb') = p.
from consis obtain consis': history-consistent θ p_2 (sb_1 @ sb')
  by (simp add: Prog sb)
from history-consistent-hd-prog-p [OF consis']
have history-consistent θ (hd-prog p (sb_1 @ sb')) (sb_1 @ sb')
  by (rule history-consistent-hd-prog)
from Cons.hyps [OF this last-prog-p]
have last-prog (hd-prog p sb_1) sb_1 = hd-prog p sb_1.
moreover
from last-prog-hd-prog-last-prog [OF last-prog-p]
have last-prog (hd-prog p sb') sb_1 = last-prog p sb_1.
ultimately
have last-prog p_2 sb_1 = hd-prog p sb'
  by simp
thus ?thesis
  by (simp add: Prog sb)
next
case Ghost sb
with Cons
show ?thesis
  by (auto split: option.splits)
qed
qed

lemma flush-all-until-volatile-write-Write sb-non-volatile-commute:
  ⋀ i m. [i < length ts; ts!i=(p,is,xs,Write sb,False a sop v A L R W#sb,D,O,R);
  ∀ i < length ts. (∀ j < length ts. i ≠ j --->
            (let (,-,-,sb_i,,-,-) = ts!i;
                (,-,-,sb_j,,-,-) = ts!j
            in outstanding-refs is-Write sb sb_i ∩ outstanding-refs is-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb_j) = {}
            ));
  ∀ j < length ts. i ≠ j --->
            (let (,-,-,sb_j,,-,-) = ts!j in a /∈ outstanding-refs is-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb_j))\]
  ⇒ flush-all-until-volatile-write (ts[i := (p,is,xs,sb,D',O,R')])(m(a := v)) =
  flush-all-until-volatile-write ts m

proof (induct ts)
case Nil thus ?case
  by simp
next
case (Cons l ts)
  note i-bound = ⟨i < length (l#ts)⟩
  note ith = ⟨(l#ts)!i = (p,is,xs,Write sb,False a sop v A L R W#sb,D,O,R)⟩
  note disj = ∀ i < length (l#ts). (∀ j < length (l#ts). i ≠ j --->
            (let (,-,-,sb_i,,-,-) = (l#ts)!i;
                (,-,-,sb_j,,-,-) = (l#ts)!j
            in outstanding-refs is-Write sb sb_i ∩ outstanding-refs is-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb_j) = {}
            )⟩


\textbf{note} \quad a \text{-} notin = \forall j < \text{length } (1 \# \text{ts}), \ i \neq j \rightarrow
\begin{align*}
(\text{let } (-,\cdots,sb_j,\cdots) = (1 \# \text{ts})!j)
\end{align*}
in \ a \not\in \text{outstanding-refs } \text{is-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_j));

\textbf{show} \quad ?\text{case}
\textbf{proof} \quad (\text{cases } i)
\begin{align*}
\text{case } 0 \quad & \text{from ith } 0 \text{ have } l = (p, is, xs, Write_{sb} \ False \ a) \text{ sop } v \ A \ L \ R \ W \# sb, D, O, R) \\
& \quad \text{by simp}
\text{thus} \quad ?\text{thesis} \\
& \quad \text{by (simp add: } 0 \ \text{del: fun-upd-apply)}
\end{align*}
\textbf{next}
\begin{align*}
\text{case } \text{(Suc } n) \quad & \text{obtain } p_l \ is_l \ D_l \ O_l \ R_l \ xs_l \ sb_l \ \text{where} \ l = (p_l, is_l, xs_l, sb_l, D_l, O_l, R_l) \\
& \quad \text{by (cases } l)
\end{align*}
\textbf{from} \quad \text{ith } \text{disj a-notin}
\textbf{have} 
\begin{align*}
& \quad \text{flush-all-until-volatile-write } (\text{ts}[n := (p, is, xs, sb, D', O', R')]) \\
& \quad \quad ((\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_l) m)(a := v)) = \\
& \quad \quad \text{flush-all-until-volatile-write } ts \\
& \quad \quad \quad \text{(flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_l) m)
\end{align*}
\textbf{apply} \quad –
\textbf{apply} \quad (\text{rule Cons.hyps})
\textbf{apply} \quad (\text{force simp add: Suc Let-def simp del: o-apply})
\textbf{done}

\textbf{moreover}
\begin{align*}
& \quad \text{let } \text{?SB} = \text{outstanding-refs } \text{is-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_l) \\
& \quad \text{have } \ a \not\in \text{?SB} \\
& \quad \text{proof} \\
& \quad \text{assume } a \in \text{?SB} \\
& \quad \text{with } a \text{-notin } [\text{rule-format, of } 0] \\
& \quad \text{show False} \\
& \quad \text{by } (\text{auto simp add: } 1 \ \text{Suc})
\end{align*}
\textbf{qed}
\textbf{then}
\textbf{have} 
\begin{align*}
& \quad ((\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_l) m)(a := v)) = \\
& \quad \quad (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_l) (m(a := v)))
\end{align*}
\textbf{apply} \quad –
\textbf{apply} \quad (\text{rule update-commute'} [\text{where } m=m \ \text{and } F={a} \ \text{and } G=?SB])
\textbf{apply} \quad (\text{auto intro:} \\
& \quad \quad \quad \text{flush-unchanged-addresses} \\
& \quad \quad \quad \text{flushed-values-mem-independent simp del: o-apply})
\textbf{done}

\textbf{ultimately}
\textbf{show} \quad ?\text{thesis} \\
\textbf{by} \quad (\text{simp add: } 1 \ \text{Suc del: fun-upd-apply o-apply})
\textbf{qed}
qed

lemma (in program) history-consistent-access-last-read':
\[ \forall p. \text{history-consistent } \theta p (\text{sb } @ [\text{Read}_{sb} \text{ volatile a t v}]) \implies \theta t = \text{Some v} \]
apply (induct sb)
apply (auto split: memref.splits option.splits)
done

lemma (in program) history-consistent-access-last-read:
\[ \text{history-consistent } \theta p (\text{rev (Read}_{sb} \text{ volatile a t v } # \text{sb})) \implies \theta t = \text{Some v} \]
by (simp add: history-consistent-access-last-read'

lemma flush-all-until-volatile-write-Read_{sb}-commute:
\[ \forall i m. [i < \text{length ts}; ts![i]=(p,\text{is},\theta,\text{Read}_{sb} \text{ volatile a t v } # \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R})]] \implies \text{flush-all-until-volatile-write } (ts[i := (p,\text{is},\theta, \text{sb, } \mathcal{D}', \mathcal{O}', \mathcal{R}'')) m] = \text{flush-all-until-volatile-write } ts m \]
proof (induct ts)
case Nil thus ?case
  by simp
next
case (Cons l ts)
note i-bound = \langle i < \text{length (l#ts)} \rangle
note ith = \langle (l#ts)!i = (p,\text{is},\theta, \text{Read}_{sb} \text{ volatile a t v } # \text{sb, } \mathcal{D}, \mathcal{O}, \mathcal{R}) \rangle:
show ?case
proof (cases i)
case 0
  from ith 0 have l: l = (p,\text{is},\theta, \text{Read}_{sb} \text{ volatile a t v } # \text{sb, } \mathcal{D}, \mathcal{O}, \mathcal{R})
  by simp
  thus ?thesis
  by (simp add: 0 del: fun-upd-apply)
next
case (Suc n)
  obtain p \_ i \_ R \_ D \_ \_ \_ sb \_ where l: l = (p,\_\_i,\_\_i,\_\_sb, \_\_D, \_\_O, \_\_R)
  by (cases l)

  from i-bound ith
  have flush-all-until-volatile-write (ts[n := (p,\_\_i,\_\_i, \_\_sb, \_\_D', \_\_O', \_\_R'\_\_)]
    (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{sb}_1 ) \text{m}) =
    \text{flush-all-until-volatile-write } ts
    (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{sb}_1 ) \text{m})
  apply 
  apply (rule Cons.hyps)
  apply (auto simp add: Suc l)
done

  then show ?thesis
  by (simp add: Suc l)
qed
qed
lemma flush-all-until-volatile-write-Ghost_commute:
\[\forall i. [i < \text{length } ts; ts!i=\langle p,\text{is},\theta,\text{Ghost}_{sb}\ A\ L\ R\ W\#sb, D, O, R \rangle]\]  
\[\implies\text{flush-all-until-volatile-write }\left(ts[i := \langle p',\text{is}',\theta', sb, D', O', R' \rangle]\right)\ m\]  
= \text{flush-all-until-volatile-write } ts\ m

proof (induct ts)
case Nil thus ?case
  by simp
next
case (Cons l ts)
  note i-bound = \langle i < \text{length } (l#ts)\rangle
  note ith = \langle (l#ts)!i = \langle p,\text{is},\theta,\text{Ghost}_{sb}\ A\ L\ R\ W\#sb, D, O, R \rangle\rangle
  show ?case
  proof (cases i)
    case 0
    from ith 0 have l: l = \langle p,\text{is},\theta,\text{Ghost}_{sb}\ A\ L\ R\ W\#sb, D, O, R \rangle
      by simp
    thus ?thesis
      by (simp add: 0 del: fun-upd-apply)
    next
    case (Suc n)
    obtain p1 l_1 O_1 R_1 D_1 sb_1 where l: l = \langle p_1,\text{is}_1,\theta_1,sb_1,D_1,O_1,R_1 \rangle
      by (cases l)
    from i-bound ith
    have \text{flush-all-until-volatile-write }\left(ts[n := \langle p',\text{is}',\theta', sb, D', O', R' \rangle]\right)
      (\text{flush }\left(\text{takeWhile }\left(\text{Not }\circ \text{is-volatile-Write}_{sb}\right)\ sb_1\right)\ m)\]  
      = \text{flush-all-until-volatile-write } ts\ m
      apply \--
      apply (rule Cons.hyps)
      apply (auto simp add: Suc l)
    done
    then show ?thesis
      by (simp add: Suc l)
  qed
qed

lemma flush-all-until-volatile-write-Prog_commute:
\[\forall i. [i < \text{length } ts; ts!i=\langle p,\text{is},\theta,\text{Prog}_{sb}\ p_1\ p_2\ mis\#sb, D, O, R \rangle]\]  
\[\implies\text{flush-all-until-volatile-write }\left(ts[i := \langle p,\text{is}, \theta, sb, D', O', R' \rangle]\right)\ m\]  
= \text{flush-all-until-volatile-write } ts\ m

proof (induct ts)
case Nil thus ?case
  by simp
next
case (Cons l ts)
  note i-bound = \langle i < \text{length } (l#ts)\rangle
  note ith = \langle (l#ts)!i = \langle p,\text{is},\theta,\text{Prog}_{sb}\ p_1\ p_2\ mis\#sb, D, O, R \rangle\rangle
  show ?case
  proof (cases i)
    case 0
    from ith 0 have l: l = \langle p,\text{is},\theta,\text{Prog}_{sb}\ p_1\ p_2\ mis\#sb, D, O, R \rangle
      by simp
    thus ?thesis
      by (simp add: 0 del: fun-upd-apply)
    next
    case (Suc n)
    obtain p1 is_1 R_1 D_1 sb_1 where l: l = \langle p_1,\text{is}_1,\theta_1,sb_1,D_1,O_1,R_1 \rangle
      by (cases l)
    from i-bound ith
    have \text{flush-all-until-volatile-write }\left(ts[n := \langle p',\text{is}',\theta', sb, D', O', R' \rangle]\right)
      (\text{flush }\left(\text{takeWhile }\left(\text{Not }\circ \text{is-volatile-Write}_{sb}\right)\ sb_1\right)\ m)\]  
      = \text{flush-all-until-volatile-write } ts\ m
      apply \--
      apply (rule Cons.hyps)
      apply (auto simp add: Suc l)
    done
    then show ?thesis
      by (simp add: Suc l)
  qed
qed
show ?case
proof (cases i)
case 0
  from ith 0 have l: l = (p, is, \theta, \text{Prog}_\text{sb} p_1 p_2 \text{mis}#sb, D, O, R)
  by simp
  thus ?thesis
    by (simp add: 0 flush-append-Prog\textsubscript{sb} del: fun-upd-apply)
next
  case (Suc n)
  obtain p l is l O l D l \theta l sb l where l: l = (p l, is l, \theta l, sb l, D l, O l, R l)
    by (cases l)
  from i-bound ith 
  have flush-all-until-volatile-write (ts[n := (p, is, \theta, sb, D', O, R')])
    (flush (takeWhile (Not \circ is-volatile-Write\textsubscript{sb}) sb l) m) =
    flush-all-until-volatile-write ts
    (flush (takeWhile (Not \circ is-volatile-Write\textsubscript{sb}) sb l) m)
  apply -
  apply (rule Cons.hyps)
  apply (auto simp add: Suc l)
  done
then show ?thesis
  by (simp add: Suc l)
qed

lemma flush-all-until-volatile-write-append-Prog\textsubscript{sb}-commute:
  \[\forall i m. [i < \text{length ts}; ts!i=(p, is, \theta, sb, D, O, R)] \implies \]
  flush-all-until-volatile-write (ts[i := (p_2, is@mis, \theta, sb@[\text{Prog}_\text{sb} p_1 p_2 \text{mis}], D', O, R')]) m =
  flush-all-until-volatile-write ts m
proof (induct ts)
case Nil thus ?case
  by simp
next
case (Cons l ts)
  note i-bound = \[ i < \text{length} (l#ts) \]
  note ith = \langle l#ts \rangle i = (p, is, \theta, sb, D, O, R)
  show ?case
  proof (cases i)
    case 0
    from ith 0 have l: l = (p, is, \theta, sb, D, O, R)
      by simp
    thus ?thesis
      by (simp add: 0 flush-append-Prog\textsubscript{sb} del: fun-upd-apply)
next
case (Suc n)
  obtain p l is l O l D l \theta l sb l where l: l = (p l, is l, \theta l, sb l, D l, O l, R l)
by (cases l)

from i-bound ith have flush-all-until-volatile-write
  (ts[n := (p₂, is@mis, θ, sb@[Progₜ p₁ p₂ mis], D', O, R')])
  (flush (takeWhile (Not o is-volatile-Writeₜ) sb) m) =
  flush-all-until-volatile-write ts
  (flush (takeWhile (Not o is-volatile-Writeₜ) sb) m)
  apply –
  apply (rule Cons.hyps)
  apply (auto simp add: Suc l)
  done

then show ?thesis
  by (simp add: Suc l)
qed

lemma (in program) history-consistent-append-Progₜ:
  assumes step: θ ⊢ p →ₚ (p', mis)
  shows history-consistent θ (hd-prog p xs) xs ⇒
  history-consistent θ (hd-prog p' (xs@[Progₜ p p' mis])) (xs@[Progₜ p p' mis])
proof (induct xs)
case Nil with step show ?case by simp
next
case (Cons x xs)
  note consis = ⟨history-consistent θ (hd-prog p (x # xs)) (x # xs)⟩
  note last = ⟨last-prog p (x#xs) = p⟩
  show ?case
  proof (cases x)
    case Writeₜ with Cons show ?thesis by (auto simp add: read-tmps-append)
  next
    case Readₜ with Cons show ?thesis by (auto split: option.splits)
  next
    case (Progₜ p₁ p₂ mis')
    from consis obtain
    step: θ |'(dom θ − read-tmps (xs@[Progₜ p p' mis]))| ⊢ p₁ →ₚ (p₂, mis') and
    consis': history-consistent θ p₂ xs
    by (auto simp add: Progₜ read-tmps-append)
    from last have last-p₂: last-prog p₂ xs = p
    by (simp add: Progₜ)
    from last-prog-to-last-prog-same [OF this]
    have last-prog'': last-prog p xs = p.
    from history-consistent-hd-prog [OF consis']
    have consis'': history-consistent θ (hd-prog p xs) xs.
    from Cons.hyps [OF this last-prog'']
    have history-consistent θ (hd-prog p' (xs@[Progₜ p p' mis]))

229
(xs @ [Progₜₜ p p' mis]).
from history-consistent-hd-prog [OF this]
have history-consistent θ (hd-prog p₂ (xs @ [Progₜₜ p p' mis]))
  (xs @ [Progₜₜ p p' mis]).
moreover
from history-consistent-hd-prog-p [OF consis']
have p₂ = hd-prog p₂ xs.
from hd-prog-last-prog-append-Progₜₜ [OF this [symmetric] last-p₂]
have hd-prog p₂ (xs @ [Progₜₜ p p' mis]) = p₂
  by simp
ultimately
have history-consistent θ p₂ (xs @ [Progₜₜ p p' mis])
  by simp
thus ?thesis
  by (simp add: Progₜₜ step)
next
case Ghostₜₜ with Cons show ?thesis by (auto)
qed
qed

primrec release :: 'a memref list ⇒ addr set ⇒ rels ⇒ rels
where
  release [] S R = R
| release (x#xs) S R =
  (case x of
    Writeₜₜ volatile - - A L R W ⇒
      (if volatile then release xs (S ∪ R − L) Map.empty
        else release xs S R)
  | Ghostₜₜ A L R W ⇒ release xs (S ∪ R − L) (augment-rels S R R)
  | - ⇒ release xs S R)

lemma augment-rels-shared-exchange: ∀a ∈ R. (a ∈ S') = (a ∈ S) ⇒ augment-rels S R R = augment-rels S' R R
apply (rule ext)
apply (auto simp add: augment-rels-def split: option.splits)
done

lemma sharing-consistent-shared-exchange:
assumes shared-eq: ∀a ∈ all-acquired sb. S' a = S a
assumes consis: sharing-consistent S O sb
shows sharing-consistent S' O sb
using shared-eq consis
proof (induct sb arbitrary: S S' O)
case Nil thus ?case by auto
next
case (Cons x sb)
show ?case
proof (cases x)
  case (Write sb volatile a sop v A L R W)
  show ?thesis
  proof (cases volatile)
    case True
    from Cons.prems obtain
      consis': sharing-consistent (S ⊕ W R ⊕ A L) (O ∪ A − R) sb and
      shared-eq: ∀ a ∈ A ∪ all-acquired sb. S′ a = S a
      by (clarsimp simp add: Write sb True )
    from shared-eq
    have shared-eq': ∀ a ∈ all-acquired sb. (S′ ⊕ W R ⊕ A L) a = (S ⊕ W R ⊕ A L) a
      by (auto simp add: augment-shared-def restrict-shared-def)
    from Cons.hyps [OF shared-eq' consis']
    have sharing-consistent (S′ ⊕ W R ⊕ A L) (O ∪ A − R) sb.
    thus ?thesis
    using A-shared-owns L-A A-R R-owns shared-eq
    by (auto simp add: Write sb True domIff)
  next
  case False with Cons show ?thesis
  by (auto simp add: Write sb)
qed

next
  case Read sb with Cons show ?thesis
  by auto
next
  case Prog sb with Cons show ?thesis
  by auto
next
  case (Ghost sb A L R W)
  from Cons.prems obtain
    consis': sharing-consistent (S ⊕ W R ⊕ A L) (O ∪ A − R) sb and
    shared-eq: ∀ a ∈ A ∪ all-acquired sb. S′ a = S a
    by (clarsimp simp add: Ghost sb )
  from shared-eq
  have shared-eq': ∀ a ∈ all-acquired sb. (S′ ⊕ W R ⊕ A L) a = (S ⊕ W R ⊕ A L) a
    by (auto simp add: augment-shared-def restrict-shared-def)
  from Cons.hyps [OF shared-eq' consis']
  have sharing-consistent (S′ ⊕ W R ⊕ A L) (O ∪ A − R) sb.
  thus ?thesis
  using A-shared-owns L-A A-R R-owns shared-eq
  by (auto simp add: Ghost sb domIff)
qed
qed
lemma release-shared-exchange:
assumes shared-eq: \( \forall a \in O \cup \text{all-acquired} \, S' a = S a \)
assumes consis: sharing-consistent \( S \, O \) sb
shows release sb \( (\text{dom} \, S') \, R = \text{release sb} \, (\text{dom} S) \, R \)
using shared-eq consis
proof (induct sb arbitrary: \( S \, S' \, O \, R \))
  case Nil thus \(?\) case by auto
next
case (Cons x sb)
show \(?\) case
proof (cases x)
case (Write_{sb} \, \text{volatile} \, a \, \text{sop} \, v \, A \, R \, W)
show \(?\) thesis
proof (cases \text{volatile})
case True
from Cons.prems obtain A-shared-owns: \( A \subseteq \text{dom} \, S \cup O \) and L-A: \( L \subseteq A \) and A-R: \( A \cap R = \{ \} \) and R-owns: \( R \subseteq O \)
consis': sharing-consistent \( (S \oplus W \, R \ominus A, L) (O \cup A - R) \) sb and
  shared-eq: \( \forall a \in O \cup A \cup \text{all-acquired} \, S' a = S a \)
by (clarsimp simp add: Write_{sb} True )
from shared-eq
  have shared-eq': \( \forall a \in O \cup A - R \cup \text{all-acquired} \, (S' \oplus W \, R \ominus A, L) a = (S \oplus W \, R \ominus A, L) a \)
  by (auto simp add: augment-shared-def restrict-shared-def)
from Cons.hyps [OF shared-eq' consis']
  have release sb \( (\text{dom} \, (S' \oplus W \, R \ominus A, L)) \, \text{Map.empty} = \text{release sb} \, (\text{dom} \, (S \oplus W \, R \ominus A, L)) \, \text{Map.empty} \).
  then show \(?\) thesis
  by (auto simp add: Write_{sb} True domIff)
next
case False with Cons show \(?\) thesis
by (auto simp add: Write_{sb})
qed
next
case Read_{sb} with Cons show \(?\) thesis
by auto
next
case Prog_{sb} with Cons show \(?\) thesis
by auto
next
case (Ghost_{sb} A L R W)
from Cons.prems obtain A-shared-owns: \( A \subseteq \text{dom} \, S \cup O \) and L-A: \( L \subseteq A \) and A-R: \( A \cap R = \{ \} \) and R-owns: \( R \subseteq O \)
consis': sharing-consistent \( (S \oplus W \, R \ominus A, L) (O \cup A - R) \) sb and
shared-eq: \( \forall a \in O \cup A \cup \text{all-acquired sb.} \ S' a = S a \)

by (clarsimp simp add: Ghost

from shared-eq

have shared-eq': \( \forall a \in O \cup A - R \cup \text{all-acquired sb.} \ (S' \oplus W R \ominus_A L) a = (S \oplus W R \ominus_A L) a \)

by (auto simp add: restrict-shared-def)

from A-shared-owns shared-eq R-owns have \( \forall a \in R. \ (a \in \text{dom } S) = (a \in \text{dom } S') \)

by (auto simp add: domIff)

from augment-rels-shared-exchange [OF this]

have \( (\text{augment-rels } (\text{dom } S') R R) = (\text{augment-rels } (\text{dom } S) R R) \).

with Cons.hyps [OF shared-eq' consis']

have release sb (dom (S \oplus W R \ominus_A L)) (augment-rels (dom S') R R) = release sb (dom (S \oplus W R \ominus_A L)) (augment-rels (dom S) R R) by simp

then show ?thesis

by (clarsimp simp add: Ghost sb domIff)

qed

lemma release-append:

\( \forall S R. \ \text{release } (sb \oplus xs) \ (\text{dom } S) R = \text{release } xs \ (\text{dom } (\text{share sb } S)) \ (\text{release sb } (\text{dom } (S)) R) \)

proof (induct sb)

case Nil thus ?case by auto

next
case (Cons x sb)

show ?case

proof (cases x)

case (Write sb volatile a sop v A L R W)

show ?thesis

proof (cases volatile)

case True

from Cons.hyps [of (S \oplus W R \ominus_A L) Map.empty]

show ?thesis

by (clarsimp simp add: Write sb True)

next
case False with Cons show ?thesis by (auto simp add: Write sb)

qed

next
case Read sb with Cons show ?thesis

by auto

next
case Progs sb with Cons show ?thesis

by auto

next
case (Ghost sb A L R W)

with Cons.hyps [of (S \oplus W R \ominus_A L) augment-rels (dom S) R R]

show ?thesis

by (clarsimp simp add: Ghost sb)

qed

233
locale xvalid-program = valid-program +

fixes valid

assumes valid-implies-valid-prog:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); \text{valid } ts] \implies \text{valid-prog } p \]

assumes valid-implies-valid-prog-hd:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); \text{valid } ts] \implies \text{valid-prog } (\text{hd-prog } p \ sb) \]

assumes distinct-load-tmps-prog-step:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); \emptyset \vdash p \rightarrow p'(is'); \text{valid } ts] \]
  \[ \implies \text{distinct-load-tmps } is' \land \]
  \[ \text{load-tmps } is' \cap \text{load-tmps } is = \{\} \land \]
  \[ \text{load-tmps } is' \cap \text{read-tmps } sb = \{\} \]

assumes valid-data-dependency-prog-step:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); \emptyset \vdash p \rightarrow p'(is'); \text{valid } ts] \]
  \[ \implies \text{data-dependency-consistent-instrs } (\text{dom } \emptyset \cup \text{load-tmps } is') \text{ is' } \land \]
  \[ \text{load-tmps } is' \cap \bigcup \text{fst ' store-sops } is' = \{\} \land \]
  \[ \text{load-tmps } is' \cap \bigcup \text{fst ' write-sops } sb = \{\} \]

assumes load-tmps-fresh-prog-step:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); \emptyset \vdash p \rightarrow p'(is'); \text{valid } ts] \]
  \[ \implies \text{load-tmps } is' \cap \text{dom } \emptyset = \{\} \]

assumes valid-sops-prog-step:
  \[ \emptyset \vdash p \rightarrow p'(is'); \text{valid-prog } p \implies \forall \text{sop } \in \text{store-sops } is'. \text{valid-sop } sop \]

assumes prog-step-preserves-valid:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); \emptyset \vdash p \rightarrow p'(is'); \text{valid } ts] \implies \]
  \[ \text{valid } (ts[i:= (p', is@is', \emptyset, sb@[\text{Prog}_{sb} p p' is'], D, O, R)]) \]

assumes flush-step-preserves-valid:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); (m, sb, O, R, S) \rightarrow f (m', sb', O', R', S'); \text{valid } ts] \implies \]
  \[ \text{valid } (ts[i:= (p, is, \emptyset, sb', D, O', R')]) \]

assumes sbh-step-preserves-valid:
  \[ i < \text{length } ts; \]
  \[ ts!i = (p, is, \emptyset, sb, D, O, R); (is, \emptyset, sb, m, D, O, R, S) \rightarrow_{sbh} (is', \emptyset, sb', m', D', O', R', S'); \]
valid \( \text{ts}}] \]
\[\Rightarrow\]
valid \((\text{ts}[i:=(p,\text{is}′,\theta′,\text{sb}′,\mathcal{D}',\mathcal{O}',\mathcal{R}′)])\)

**Lemma refl**: \( x = y \Rightarrow r^{**} x y \)
by auto

**Lemma no-volatile-Read\(sb\)-volatile-reads-consistent**: 
\(\forall m. \text{outstanding-refs is-volatile-Read}\(sb\) \(sb = \{\} \Rightarrow \text{volatile-reads-consistent m sb}\)

**Theorem (in program) flush-store-buffer-append**: 
**shows** \(\forall ts p m \ \emptyset \mathcal{O} \mathcal{R} \mathcal{D} \mathcal{S} \text{is } \mathcal{O}'\). 
\([i < \text{length ts}; \]
\(\text{instrs } (sb\oplus sb') \oplus \text{is}_{sb} = \text{is} \oplus \text{prog-instrs} (sb\oplus sb');\)
\(\text{causal-program-history is}_{sb} (sb\oplus sb');\)
\(\text{tsli} = (p,\text{is},\emptyset | (\text{dom } \emptyset - \text{read-tmps} (sb\oplus sb'))),x,\mathcal{D},\mathcal{O},\mathcal{R});\)
\(p=\text{hd-prog } p_{sb} (sb\oplus sb');\)
\(\text{(last-prog } p_{sb} (sb\oplus sb')) = p_{sb};\)
\(\text{reads-consistent True } \mathcal{O}' m sb;\)
\(\text{history-consistent } \emptyset p (sb\oplus sb');\)
\(\forall \text{sop } \in \text{write-sops sb}. \text{valid-sop sop};\)
\(\text{distinct-read-tmps } (sb\oplus sb');\)
\(\text{volatile-reads-consistent m sb}\)
\(] \)
\[\Rightarrow\]
\(\exists \text{is}'. \text{instrs } sb' \oplus \text{is}_{sb} = \text{is}' \oplus \text{prog-instrs} sb' \land\)
\((\text{ts,m},\mathcal{S}) \Rightarrow d^*\)
\((\text{ts}[i:=(\text{last-prog } (\text{hd-prog } p_{sb} sb') \text{sb},\text{is}',\emptyset | (\text{dom } \emptyset - \text{read-tmps sb'}),x,\mathcal{D} \lor \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\})),\)
\(\text{acquired True sb } \emptyset, \text{release sb } (\text{dom } \mathcal{S} ) \mathcal{R}),\text{flush sb m,share sb } \mathcal{S}\)

**Proof (induct sb)**
**Case** Nil
**Thus** ?\text{case by } (auto simp add: list-update-id' split: if-split-asm)

**Next**
**Case** (Cons \(r sb\))
**Interpret** direct-computation:
\(\text{computation direct-memop-step empty-storebuffer-step program-step } \lambda p \ p' \text{is sb. sb}.\)
**Have** ts-i:
\[ tsl! = (p, is, \emptyset \mid (\text{dom } \emptyset - \text{read-tmps } ((r\#sb)@sb')), x, D, O, R) \]

by fact
have is: \text{instrs } ((r \# sb) @ sb') @ is_{sb} = is @ \text{prog-instrs } ((r \# sb) @ sb') by fact

have i-bound: \( i < \text{length } ts \) by fact
have causal: \( \text{causal-program-history } is_{sb} ((r \# sb) @ sb') by fact \)

hence causal': \( \text{causal-program-history } is_{sb} (sb @ sb') \)
by (auto simp add: causal-program-history-def)

note reads-consis = \( \langle \text{reads-consistent } True \ O' \ m \ (r\#sb) \rangle \)
note p = \( \langle p = \text{hd-prog } p_{sb} ((r\#sb)@sb') \rangle \)
note p_{sb} = \( \langle \text{last-prog } p_{sb} ((r \# sb) @ sb') = p_{sb} \rangle \)
note hist-consis = \( \langle \text{history-consistent } \emptyset p ((r\#sb)@sb') \rangle \)
note valid-sops = \( \langle \forall \text{sop } \in \text{write-sops } (r\#sb), \text{valid-sop } \text{sop} \rangle \)
note dist = \( \langle \text{distinct-read-tmps } ((r\#sb)@sb') \rangle \)
note vol-read-consis = \( \langle \text{volatile-reads-consistent } m \ (r\#sb) \rangle \)

show ?case
proof (cases r)
  case (Prog sb p1 p2 pis)

  from vol-read-consis
  have vol-read-consis': \( \text{volatile-reads-consistent } m \ sb \)
  by (auto simp add: Prog sb)

  from hist-consis obtain
  prog-step: \( \emptyset | (\text{dom } \emptyset - \text{read-tmps } (sb @ sb')) \vdash p_1 \rightarrow_p (p_2, pis) \) and
  hist-consis': \( \text{history-consistent } \emptyset p_2 (sb @ sb') \)
  by (auto simp add: Prog sb)

  from p obtain \( p: p = p_1 \)
  by (simp add: Prog sb)

  from history-consistent-hd-prog [OF hist-consis']
  have hist-consis'': \( \text{history-consistent } \emptyset (\text{hd-prog } p_2 (sb @ sb')) (sb @ sb') \).

  from is
  have is: \( \text{instrs } (sb @ sb') @ is_{sb} = (is @ pis) @ \text{prog-instrs } (sb @ sb') \)
  by (simp add: Prog sb)

  from ts-i is have
  ts-i: \( tsl! = (p, is, \emptyset \mid (\text{dom } \emptyset - \text{read-tmps } (sb @ sb'))), x, D, O, R) \)
  by (simp add: Prog sb)

  let \?ts' = ts[i:= (p2, is@pis, \emptyset \mid (\text{dom } \emptyset - \text{read-tmps } (sb @ sb')), x, D, O, R)]
  from direct-computation.Program [OF i-bound ts-i prog-step [simplified p[symmetric]]]
  have \( (ts,m,\emptyset) \Rightarrow_d (?ts',m,\emptyset) \) by simp

  also
  from i-bound have i-bound': \( i < \text{length } ?ts' \)

  236
by auto

from i-bound
have ts'\text{-}i: ?ts'\text{-}i = (p_2, is@pis,(\theta \vdash (\text{dom } \theta - \text{read-tmps (sb @ sb')})),x, D, \mathcal{O}, \mathcal{R})
  by auto

from history-consistent-hd-prog-p [OF hist-consis′]
have p_2-hd-prog: p_2 = hd-prog p_2 (sb @ sb').

from reads-consis have reads-consis′: reads-consistent True \mathcal{O'} m sb
  by (simp add: Prog sb)

from valid-sops have valid-sops′: \forall sop \in \text{write-sops sb}. \text{valid-sop sop}
  by (simp add: Prog sb)

from dist have dist′: distinct-read-tmps (sb@sb')
  by (simp add: Prog sb)

from p_{sb} have last-prog-p_2: last-prog p_2 (sb @ sb') = p_{sb}
  by (simp add: Prog sb)
from hd-prog-last-prog-end [OF p_2-hd-prog this]
have p_2-hd-prog′: p_2 = hd-prog p_{sb} (sb @ sb').
from last-prog-p_2 [symmetric] have last-prog′: last-prog p_{sb} (sb @ sb') = p_{sb}
  by (simp add: last-prog-idem)

from Cons.\text{hyps} [OF i-bound′ is causal′ ts′-i p_2-hd-prog′ last-prog′ reads-consis′
hist-consis′ valid-sops′ dist′ vol-read-consis′] i-bound
obtain is′ where
  is′: instrs sb′ @ is_{sb} = is′ @ prog-instrs sb′ and
  step: (?ts′, m,S) \Rightarrow_d^∗
  (ts′ := (last-prog (hd-prog p_{sb} sb') sb, is′,
   \theta \vdash (\text{dom } \theta - \text{read-tmps sb'}), x, D \lor \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\},
   \text{acquired True sb} \mathcal{O}, \text{release sb} (\text{dom S}) \mathcal{R}),
   \text{flush sb m, share sb S})
  by (auto)
from p_2-hd-prog′
have last-prog-eq: last-prog (hd-prog p_{sb} sb') sb = last-prog p_2 sb
  by (simp add: last-prog-hd-prog-append)

note step
finally show ?thesis
  using is′
  by (simp add: Prog_{sb} last-prog-eq)

next
case (Write_{sb} volatile a sop v A L R W)
obtain D \text{ f where} sop: sop=(D,f)
  by (cases sop)
from vol-read-consis
have vol-read-consis' : volatile-reads-consistent (m(a:=v)) sb
  by (auto simp add: Write sb)

from hist-consis obtain
  D-tmps: D ⊆ dom ⊸ and
  f-v: f ⊸ = v and
  dep: D ∩ read-tmps (sb@sb') = {} and
  hist-consis': history-consistent ⊸ p (sb@sb')
  by (simp add: Write sb) split: option.splits)

from dist have dist': distinct-read-tmps (sb@sb') by (auto simp add: Write sb)

from valid-sops obtain valid-sop sop and
  valid-sops': ∀ sop ∈ write-sops sb. valid-sop sop
  by (simp add: Write sb) split: option.
  splits)

interpret valid-sop sop by fact
from valid-sop [OF sop D-tmps]
have f ⊸ = f (θ |' D).
moreover from dep D-tmps have D-subset: D ⊆ (dom ⊸ − read-tmps (sb@sb'))
  by auto
moreover from D-subset have (θ |' (dom ⊸ − read-tmps (sb@sb'))) |' D) = ⊸ |' D
  apply −
  apply (rule ext)
  apply (auto simp add: restrict-map-def)
  done
moreover from D-subset D-tmps have D ⊆ dom (θ |' (dom ⊸ − read-tmps (sb@sb')))
  by simp
moreover
note valid-sop [OF sop this]
ultimately have f-v': f (θ |' (dom ⊸ − read-tmps (sb@sb'))) = v
  by (simp add: f-v)

interpret causal': causal-program-history is sb sb@sb' by fact

from is have Write volatile a sop A L R W# instrs (sb @ sb') @ is sb = is @ prog-instrs (sb @ sb')
  by (simp add: Write sb)
with causal'.causal-program-history [of [], simplified, OF refl]
obtain is' where is: is=Write volatile a sop A L R W#is' and
  is': instrs (sb @ sb') @ is sb = is' @ prog-instrs (sb @ sb')
  by auto

from ts-i is
have ts-i: tсли = (p,Write volatile a sop A L R W#is',
  θ |' (dom ⊸ − read-tmps (sb@sb')), x,D,O,R)
  by (simp add: Write sb)
from p have p: p = hd-prog p sb\ (sb@sb)'
  by (auto simp add: Write sb hd-prog-idem)

from p sb have p sb': last-prog p sb (sb @ sb) = p sb
  by (simp add: Write sb)

show ?thesis
proof (cases volatile)
case False
  have memop-step: (Write volatile a sop A L R W#is',θ| (dom θ − read-tmps (sb@sb)'), x,m,D,O,R,S) → (is',θ| (dom θ − read-tmps (sb@sb)'),x,m(a:=v),D,O,R,S)
using D-subset
apply (simp only: sop f-v [symmetric] False)
apply (rule direct-memop-step. WriteNonVolatile)
done

let ?ts' = ts[i := (p, is',θ | | (dom θ − read-tmps (sb@sb)'), x, D, O,R)]
from direct-computation.Memop [OF i-bound ts-i memop-step]
have (ts, m, S) ⇒ d (?ts', m(a := v), S).
also
from reads-consis have reads-consis': reads-consistent True O' (m(a:=v)) sb
by (auto simp add: Write sb False)
from i-bound have i-bound': i < length ?ts'
by auto

from i-bound
have ts'! i: ?ts'! i = (p, is',θ | | (dom θ − read-tmps (sb@sb)'), x, D, O,R)
by simp
from Cons,hyps [OF i-bound' is' causal' ts'! i p' p sb' reads-consis' hist-consis' valid-sops' dist' vol-read-consis'] i-bound
obtain is'' where
is'': instrs sb' @ is_{sb'} = is'' @ prog-instrs sb' and
steps: (?ts',m(a:=v),S) ⇒ d'*
(ts[i := (last-prog (hd-prog p sb sb'), sb, is''),
  θ | | (dom θ − read-tmps sb'), x,
  D ∨ outstanding-refs is-volatile-Write sb sb ≠ {}, acquired True sb O, release sb
  (dom S) R)],
  flush sb (m(a := v)),share sb S)
by (auto simp del: fun-upd-apply)
note steps
finally
show ?thesis
using is''
by (simp add: Write sb False)
next
case True
\textbf{have} memop-step:

\[(\text{Write volatile a sop } A \mid R \mid W \# \text{is'} \vartheta \mid (\text{dom } \vartheta - \text{read-tmps } (sb @ sb'))),
\]
\[x,m, D, O, R, S) \rightarrow
\]
\[(\text{is'}, \vartheta \mid (\text{dom } \vartheta - \text{read-tmps } (sb @ sb')), x, m(a := v), \text{True}, O \cup A - R, \text{Map.empty}, S \oplus W R \ominus A L)\]

\textbf{using} D-subset

\textbf{apply} (simp only: sop f-vf [symmetric] True)

\textbf{apply} (rule direct-memop-step.\textit{WriteVolatile})

\textbf{done}

let \(?ts' = ts[i := (p, is'), \vartheta \mid (\text{dom } \vartheta - \text{read-tmps } (sb @ sb'))], x, \text{True}, O \cup A - R, \text{Map.empty})

\textbf{from} direct-computation.\textit{Memop} [OF i-bound ts-i memop-step]

\textbf{have} (ts, m, S) \Rightarrow_d (?ts', m(a := v), S \oplus W R \ominus A L).

also

\textbf{from} reads-consis \textbf{have} reads-consis': reads-consistent True (O' \cup A - R)(m(a := v))

\textbf{sb by} (auto simp add: \textit{Write_sb} True)

\textbf{from} i-bound \textbf{have} i-bound': i < length ?ts'

\textbf{by} auto

\textbf{from} i-bound

\textbf{have} ts'-i: ?ts' ! i = (p, is', \vartheta \mid (\text{dom } \vartheta - \text{read-tmps } (sb @ sb'))], x, \text{True}, O \cup A - R, \text{Map.empty})

\textbf{by} simp

\textbf{from} Cons.hyps [OF i-bound' is' causal' ts'-i p' p_{sb'} reads-consis' hist-consis'
valid-sops' dist' vol-read-consis', of (S \oplus W R \ominus A L)] i-bound

\textbf{obtain} is'' where

is'': instrs sb' @ is_{sb'} = is'' @ prog-instrs sb' and
steps: (?ts', m(a := v), S \oplus W R \ominus A L) \Rightarrow_d is''

\textbf{by} (auto simp del: fun-upd-apply)

\textbf{note} steps

\textbf{finally}

\textbf{show} ?thesis

\textbf{using} is''

\textbf{by} (simp add: \textit{Write_sb} True)

\textbf{qed}

\textbf{next}

\textbf{case} (Read_{sb} volatile a t v)

\textbf{from} vol-read-consis \textbf{obtain} v: v=m a and r-consis: reads-consistent True O' m sb and

\textbf{by} (cases volatile) (auto simp add: \textit{Read_sb})
from valid-sops have valid-sops: \( \forall \text{sop} \in \text{write-sops sb} \) valid-sop \( \text{sop} \)
by (simp add: Read sb)

from hist-consis obtain \( \vartheta \) t = Some v and hist-consis: history-consistent \( \vartheta \) p (sb@sb')
by (simp add: Read sb split: option.splits)
from dist obtain t-notin: \( t \notin \text{read-tmps (sb@sb')} \) and dist': distinct-read-tmps (sb@sb')
by (simp add: Read sb)

interpret causal': causal-program-history is sb sb@sb'
by fact

interpret causal': causal-program-history
with causal': causal-program-history [of [], simplified, OF refl]
obtain is' where is: is=Read volatile a t#is' and is': instrs (sb @ sb') @ is sb = is' @ prog-instrs (sb @ sb')
by auto

from ts-i is have ts-i: tsli = (p,Read volatile a t#is',
\( \vartheta \) t (dom \( \vartheta \) - insert t (read-tmps (sb@sb'))),x,D,O,R)
by (simp add: Read sb)

from direct-memop-step.Read [of volatile a t is' \( \vartheta \) t (read-tmps (sb@sb'))] x m D O R S]
have memop-step:
(Read volatile a t # is',
\( \vartheta \) t (dom \( \vartheta \) - insert t (read-tmps (sb@sb'))), x, m, D, O, R, S) \( \to \)
(is',
\( \vartheta \) t (dom \( \vartheta \) - (read-tmps (sb@sb'))), x, m, D, O, R, S)
by (simp add: v [symmetric] restrict-commute restrict-commute')
let \( \textit{ts}' = \textit{ts}[i := (p, \textit{is}', \textit{θ}|\textit{read-tmps (sb @ sb')}), \textit{x}, \textit{D}, \textit{O}, \textit{R}] \)

from direct-computation.Memop [OF i-bound ts-i memop-step]

have \( (\textit{ts}, \textit{m}, \textit{S}) \Rightarrow_d (\textit{ts}'', \textit{m}, \textit{S}) \).

also

from i-bound have i-bound': \( i < \text{length } \textit{ts}' \)

by auto

from i-bound

have \( \textit{ts}'i: \textit{ts}'i = (p, \textit{is}', (\emptyset|\textit{read-tmps (sb @ sb')}), \textit{x}, \textit{D}, \textit{O}, \textit{R}) \)

by auto

from \( \textit{p} \) have \( \textit{p}' : \textit{p} = \text{hd-prog } \textit{p}_{\textit{sb}} (\textit{sb} @ \textit{sb'}) \)

by (auto simp add: Read sb hd-prog-idem)

from \( \textit{p}_{\textit{sb}} \) have \( \textit{p}_{\textit{sb}}': \text{last-prog } \textit{p}_{\textit{sb}} (\textit{sb} @ \textit{sb'}) = \textit{p}_{\textit{sb}} \)

by (simp add: Read sb)

from Cons.hyps [OF i-bound' is' causal' ts' i p' p_{\textit{sb}}' r-consis hist-consis' valid-sops' dist' vol-read-consis']

obtain \( \textit{is}'' \) where

\( \textit{is}'' : \text{instrs } \textit{sb}' @ \textit{is}_{\textit{sb}} = \text{instrs } \textit{sb}' @ \text{prog-instrs } \textit{sb}' \) and

steps: \( (?\textit{ts}'i, \textit{m}, \textit{S}) \Rightarrow_d \)

\( (\textit{ts}[i := (\text{last-prog } (\text{hd-prog } \textit{p}_{\textit{sb}} \textit{sb'})) \textit{sb}, \textit{is}'', \emptyset|\textit{read-tmps sb'}, \textit{x}, \textit{D} \lor \text{outstanding-refs is-volatile-Write}_{\textit{sb}} \textit{sb} \neq \{\}, \)

acquired True \( \textit{sb} \textit{O}, \text{release } \textit{sb} (\text{dom } \textit{S} \textit{R})], \)

flush \( \textit{sb} \textit{m}, \text{share } \textit{sb} \textit{S}) \)

by (auto simp del: fun-upd-apply)

note steps

finally

show \( ?\text{thesis} \)

using \( \textit{is}'' \)

by (simp add: Read sb)

next

case (\text{Ghost}_{\textit{sb}} A L R W)

from vol-read-consis

have \( \text{vol-read-consis'}; \text{volatile-reads-consistent } \textit{m sb} \)

by (auto simp add: \text{Ghost}_{\textit{sb}})

from reads-consis have \( \text{r-consis: reads-consistent True } (\textit{O'} \cup A - R) \textit{m sb} \)

by (auto simp add: \text{Ghost}_{\textit{sb}})
from valid-sops have valid-sops': \( \forall \text{sop} \in \text{write-sops} \) \( \text{sb} \). valid-sop \( \text{sop} \)
by (simp add: Ghost\( \text{sb} \))

from hist-consis obtain
  hist-consis': history-consistent \( \theta \) \( p \) (sb@sb')
by (simp add: Ghost\( \text{sb} \))

from dist obtain
dist': distinct-read-tmps (sb@sb') by (simp add: Ghost\( \text{sb} \))

interpret causal': causal-program-history is\( _{sb} \) sb@sb' by fact

from is
have Ghost A L R W# instrs (sb @ sb') @ is\( _{sb} \) = is @ prog-instrs (sb @ sb')
by (simp add: Ghost\( \text{sb} \))

with causal'.causal-program-history [of [], simplified, OF refl]
obtain is': where is: is=Ghost A L R W#is' and
  is': instrs (sb @ sb') @ is\( _{sb} \) = is' @ prog-instrs (sb @ sb')
by auto

from ts-i is
have ts-i: ts\( !i \) = (p, Ghost A L R W#is',
  \( \theta \mid \cdot (\text{dom} \ \theta - \text{(read-tmps (sb@sb'))}) \), x, D, O, R)
by (simp add: Ghost\( \text{sb} \))

from direct-memop-step.Ghost [of A L R W is']
  \( \theta \mid \cdot (\text{dom} \ \theta - \text{(read-tmps (sb@sb'))}) \) \( x \) \( m \) D O R S]
have memop-step:
  (Ghost A L R W# is',\( \theta \mid \cdot (\text{dom} \ \theta - \text{(read-tmps (sb@sb'))}) \), x, m, D, O, R, S)
  \( \rightarrow (is',\theta \mid \cdot (\text{dom} \ \theta - \text{(read-tmps (sb@sb'))}) \), x, m, D, O \cup A - R , \text{augment-rels (dom } S) \) \( R \ R', S \oplus W \ R \ominus A L \).

let ?ts' = ts\( !i \) := (p, is',
  \( \theta \mid \cdot (\text{dom} \ \theta - \text{(read-tmps (sb@sb'))}) \), x, D, O \cup A - R , \text{augment-rels (dom } S) \) \( R \ R', S \oplus W \ R \ominus A L \)]
from direct-computation.Memop [OF i-bound ts-i memop-step]
have (ts, m, S) \( \Rightarrow_{d} (\ ?ts' , m, S \oplus W \ R \ominus A L \).

also

from i-bound have i-bound': i < length ?ts'
by auto

from i-bound
have ts'!i: ?ts'\!i = (p, is',(\( \theta \mid \cdot (\text{dom} \ \theta - \text{(read-tmps (sb@sb'))}) \)), x, D, O \cup A - R,\text{augment-rels (dom } S) \) \( R \ R' \)
by auto
from p have p': p = hd-prog p\textsubscript{sb} (sb@sb')
  by (auto simp add: Ghost\textsubscript{sb} hd-prog-idem)

from p\textsubscript{sb} have p\textsubscript{sb}': last-prog p\textsubscript{sb} (sb \@ sb') = p\textsubscript{sb}
  by (simp add: Ghost\textsubscript{sb})

from Cons.hyps [OF i-bound' is' causal' ts'i p' p\textsubscript{sb}' r-consis hist-consis'
  valid-sops' dist' vol-read-consis', of S ↪ W R ↪ \mathcal{A} L]
obtain is'' where
  is'': instrs sb' @ is\textsubscript{sb} = is'' @ prog-instrs sb' and
  steps: (?ts'i,m,S ↪ W R ↪ \mathcal{A} L) ⇒ d\ast
    (ts[i := (last-prog (hd-prog p\textsubscript{sb} sb') sb, is''),
     \varnothing | (dom \varnothing − read-tmps sb)\textsubscript{sb'},x,
     D ∨ outstanding-refs is-volatile-Write\textsubscript{sb} sb ≠ \{\}, acquired True sb (\mathcal{O} \cup A − R),
     release sb (dom (S ↪ W R ↪ \mathcal{A} L)) (augment-rels (dom S) R \mathcal{R})],
     flush sb m, share sb (S ↪ W R ↪ \mathcal{A} L))
by (auto simp add: list-update-overwrite simp del: fun-upd-apply)

note steps
finally
show ?thesis
  using is''
  by (simp add: Ghost\textsubscript{sb})
qed

lemma last-prog-same-append: \(\forall xs\) p\textsubscript{sb}. last-prog p\textsubscript{sb} (sb@xs) = p\textsubscript{sb} \implies last-prog p\textsubscript{sb} xs = p\textsubscript{sb}
apply (induct sb)
apply simp
subgoal for a sb xs psb
apply (case-tac a)
apply simp
apply simp
apply simp
apply (drule last-prog-to-last-prog-same)
apply simp
apply simp
done
done

lemma reads-consistent-drop-volatile-writes-no-volatile-reads:
\( \forall \text{pending-write } O, m. \text{read-consistent pending-write } O, m, \text{sb} \implies \text{outstanding-refs is-volatile-Read}_{sb} ((\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}) = \{\}) \\
\text{apply (induct sb)}
\text{apply (auto split: memref.splits)}
done

lemma reads-consistent-flush-other:
assumes no-volatile-Write_{sb}: \text{outstanding-refs is-volatile-Write}_{sb} \text{ sb} = \{\}
shows \( \forall m \text{ pending-write } O. \) \text{outsanding-refs is-volatile-Read}_{sb} \text{ sb} = \{\}:
\text{read-consistent pending-write } O, m, \text{xs} \implies \text{read-consistent pending-write } O (\text{flush sb m}) \text{ xs}
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
note no-inter = \( \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb} = \{\}\)
hence no-inter': \text{outsanding-refs is-non-volatile-Read}_{sb} \text{ xs} \implies \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb} = \{\}
by (auto)
note consis = \( \text{read-consistent pending-write } O, m, (x \neq \text{ xs})\)
show ?case
proof (cases x)
case (Write_{sb} volatile a sop v A L R)
show ?thesis
proof (cases volatile)
case False
from consis obtain consis': \text{read-consistent pending-write } O (m(a := v)) \text{ xs}
by (simp add: Write_{sb} False)
from Cons.hyps [OF no-inter' consis']
have reads-consistent pending-write \( O \) (flush sb (m(a := v))) xs.

moreover
from no-inter have a \( \notin \) outstanding-refs is-non-volatile-Write\(sb\) sb
by (auto simp add: Write\(sb\) split: if-split-asm)

from flush-update-other' [OF this no-volatile-Write\(sb\)-sb]
have (flush sb (m(a := v))) = (flush sb m)(a := v).

ultimately
show \(?thesis\)
by (simp add: Write\(sb\) False)

next
case True
from consis obtain consis': reads-consistent True (\( O \cup A - R \)) (m(a := v)) xs and
no-read: (outstanding-refs is-volatile-Read\(sb\) xs = \{\})
by (simp add: Write\(sb\) True)

from Cons.hyps [OF no-inter' consis']
have reads-consistent True (\( O \cup A - R \)) (flush sb (m(a := v))) xs.

moreover
from no-inter have a \( \notin \) outstanding-refs is-non-volatile-Write\(sb\) sb
by (auto simp add: Write\(sb\) split: if-split-asm)

from flush-update-other' [OF this no-volatile-Write\(sb\)-sb]
have (flush sb (m(a := v))) = (flush sb m)(a := v).

ultimately
show \(?thesis\)
using no-read
by (simp add: Write\(sb\) True)
qed

next
case (Read\(sb\) volatile a t v)
from consis obtain val: (\( \neg \) volatile \( \rightarrow \) (pending-write \( \lor \) a \( \in \) \( O \)) \( \rightarrow \) v = m a) and
consis': reads-consistent pending-write \( O \) m xs
by (simp add: Read\(sb\))
from Cons.hyps [OF no-inter' consis']
have hyp: reads-consistent pending-write \( O \) (flush sb m) xs
by simp
show \(?thesis\)
proof (cases volatile)
case False
from no-inter False have a \( \notin \) outstanding-refs is-non-volatile-Write\(sb\) sb
by (auto simp add: Read\(sb\) split: if-split-asm)
with no-volatile-Write\(sb\)-sb
have a \( \notin \) outstanding-refs is-Write\(sb\) sb
apply (clarsimp simp add: outstanding-refs-conv is-Write\(sb\)-def split: memref.splits)
apply force
done
with hyp val flush-unchanged-addresses [OF this]
show \(?thesis\)
by (simp add: Read\(sb\))
next

246
case True
  with hyp val show ?thesis
by (simp add: Read_sb)
qed
next
  case Progsb with Cons show ?thesis by auto
next
  case Ghostsb with Cons show ?thesis by auto
qed

lemma reads-consistent-flush-independent:
  assumes no-volatile-Write\_sb: outstanding-refs is-Write\_sb sb ∩ outstanding-refs
  is-non-volatile-Read\_sb xs = {}
  assumes consis: reads-consistent pending-write \( O \) m xs
  shows reads-consistent pending-write \( O \) (flush sb m) xs
proof –
  from flush-unchanged-addresses [where sb=sb and m=m] no-volatile-Write\_sb
  have ∀a ∈ outstanding-refs is-non-volatile-Read\_sb xs. flush sb m a = m a
    by auto
  from reads-consistent-mem-eq-on-non-volatile-reads [OF this subset-refl consis]
  show ?thesis .
qed

lemma reads-consistent-flush-all-until-volatile-write-aux:
  assumes no-reads: outstanding-refs is-volatile-Read\_sb xs = {}
  shows ⋀ m pending-write \( O' \). [reads-consistent pending-write \( O' \) m xs; ∀i < length ts.
    let (p, is, t, O, R) = ts!i in
    outstanding-refs (Not ◦ is-volatile-Read\_sb) xs ∩
    outstanding-refs is-non-volatile-Write\_sb (takeWhile (Not ◦ is-volatile-Write\_sb) sb) = {}
    ]
  ⇒ reads-consistent pending-write \( O' \) (flush-all-until-volatile-write ts m) xs
proof (induct ts)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  have consis: reads-consistent pending-write \( O' \) m xs by fact

obtain p_t is_t O_t D_t t D_t sb_t
  where t: t=(p_t,is_t,t,D_t,O_t,R_t)
  by (cases t)

from Cons.prems t obtain
  no-inter: outstanding-refs (Not ◦ is-volatile-Read\_sb) xs ∩
  outstanding-refs is-non-volatile-Write\_sb (takeWhile (Not ◦ is-volatile-Write\_sb) sb_t) = {}
  and
  no-inter': ∀i < length ts.
  let (p,is,t,db,O,R) = ts!i in

247
outstanding-refs (Not ◦ is-volatile-Read_{sb}) xs ∩ outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) = 
{}
by (force simp add: Let-def simp del: o-apply)

have out1: outstanding-refs is-volatile-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_t) = 
{}
by (auto simp add: outstanding-refs-conv dest: set-takeWhileD)

from no-inter have outstanding-refs (Not ◦ is-volatile-Read_{sb}) xs ∩ outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_t) = 
{}
by auto

from reads-consistent-flush-other [OF out1 this consis] have reads-consistent pending-write O' (flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_t) m) xs.
from Cons.hyps [OF this no-inter'] show ?case
  by (simp add: t)
qed

lemma reads-consistent-flush-other':
assumes no-volatile-Write_{sb}-sb: outstanding-refs is-volatile-Write_{sb} sb = {}
shows \( \bigwedge m. O. \)
[outstanding-refs is-non-volatile-Write_{sb} sb ∩ 
(outstanding-refs is-volatile-Write_{sb} xs ∪ 
  outstanding-refs is-non-volatile-Write_{sb} xs ∪ 
  outstanding-refs is-non-volatile-Read_{sb} (dropWhile (Not ◦ is-volatile-Write_{sb}) xs)
∪ 
  (outstanding-refs is-non-volatile-Read_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) xs)
− RO) ∪ 
  (O ∪ all-acquired (takeWhile (Not ◦ is-volatile-Write_{sb}) xs))
) = 
{};
reads-consistent False O m xs;
read-only-reads O (takeWhile (Not ◦ is-volatile-Write_{sb}) xs) ⊆ RO]
==\Rightarrow reads-consistent False O (flush sb m) xs
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)

note no-inter = Cons.prems (1)

note consis = (reads-consistent False O m (x # xs))
have aargh: (Not ◦ is-volatile-Write_{sb}) = (λa. ¬ is-volatile-Write_{sb} a)
by (rule ext) auto

note RO = \langle read-only-reads O (takeWhile (Not ∘ is-volatile-Write sb) (x#xs)) \subseteq RO \rangle

show ?case
proof (cases x)
  case (Write sb volatile a sop v A L R)
  show ?thesis
  proof (cases volatile)
  case False
  from consis obtain consis': reads-consistent False O (m(a := v)) xs
  by (simp add: Write sb False)
  from no-inter
  have no-inter': outstanding-refs is-non-volatile-Write sb sb ∩
    (outstanding-refs is-volatile-Write sb xs ∪
      outstanding-refs is-non-volatile-Write sb xs ∪
      outstanding-refs is-non-volatile-Read sb (dropWhile (Not ∘ is-volatile-Write sb) xs)
    )
  − RO ∪
    (O ∪ all-acquired (takeWhile (Not ∘ is-volatile-Write sb) xs))
  = {}
  by (clarsimp simp add: Write sb False split: if-split-asm)
  from RO
  have RO': read-only-reads O (takeWhile (Not ∘ is-volatile-Write sb) xs) \subseteq RO
  by (auto simp add: Write sb False)
  from Cons.hyps [OF no-inter' consis' RO']
  have reads-consistent False O (flush sb (m(a := v))) xs.
  moreover
  from no-inter have a \notin outstanding-refs is-non-volatile-Write sb sb
  by (auto simp add: Write sb split: if-split-asm)
  from flush-update-other' [OF this no-volatile-Write sb]
  have (flush sb (m(a := v))) = (flush sb m)(a := v).
  ultimately
  show ?thesis
  by (simp add: Write sb False)
next
  case True
  from consis obtain consis': reads-consistent True (O ∪ A − R) (m(a := v)) xs and
  no-read: (outstanding-refs is-volatile-Read sb xs = {})
  by (simp add: Write sb True)
  from no-inter obtain
a-notin: a \notin \text{outstanding-refs} \quad \text{is-non-volatile-Write}_{sb} \quad \text{and}

disj: (\text{outstanding-refs} (\text{Not} \circ \text{is-volatile-Read}_{sb}) \, \text{xs}) \cap \text{outstanding-refs} \, \text{is-non-volatile-Write}_{sb} \, \text{sb} = \{\}

\text{by} \quad (\text{auto simp add: Write}_{sb} \, \text{True} \, \text{aargh} \, \text{misc-outstanding-refs-convs})

\text{from reads-consistent-flush-other} \quad [\text{OF no-volatile-Write}_{sb} \, \text{sb} \, \text{disj} \, \text{consis}^\dagger]

\text{have} \quad \text{reads-consistent True} \quad (O \cup A - R) \, (\text{flush sb} \, (m(a := v))) \, \text{xs}.

\text{moreover}

\text{note} \quad \text{a-notin}

\text{from flush-update-other}^\prime \quad [\text{OF this no-volatile-Write}_{sb} \, \text{sb}]

\text{have} \quad (\text{flush sb} \, (m(a := v))) = (\text{flush sb} \, (m(a := v)).

\text{ultimately}

\text{show} \quad \text{?thesis}

\text{using} \quad \text{no-read}

\text{by} \quad (\text{simp add: Write}_{sb} \, \text{True})

\text{qed}

\text{next}

case \quad (\text{Read}_{sb} \, \text{volatile a t v})

\text{from consis} \quad \text{obtain} \quad \text{val:} \quad (\neg \text{volatile} \rightarrow a \in O \rightarrow v = m \, a) \quad \text{and}

\text{consis}^\dagger: \quad \text{reads-consistent False} \quad O \quad m \quad \text{xs}

\text{by} \quad (\text{simp add: Read}_{sb})

\text{from RO}

\text{have} \quad \text{RO}^\dagger: \quad \text{read-only-reads} \quad O \quad (\text{takeWhile} \, (\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{xs}) \subseteq RO

\text{by} \quad (\text{auto simp add: Read}_{sb} \, \text{True})

\text{from no-inter}

\text{have} \quad \text{no-inter}^\dagger: \quad \text{outstanding-refs} \, \text{is-non-volatile-Write}_{sb} \, \text{sb} \cap

(\text{outstanding-refs} \, \text{is-volatile-Write}_{sb} \, \text{xs} \cup
\text{outstanding-refs} \, \text{is-non-volatile-Write}_{sb} \, \text{xs} \cup
\text{outstanding-refs} \, \text{is-non-volatile-Read}_{sb} \, \text{dropWhile} \, (\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{xs}) \cup
(\text{outstanding-refs} \, \text{is-non-volatile-Read}_{sb} \, \text{takeWhile} \, (\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{xs}) - RO \cup
(O \cup \text{all-acquired} \, \text{takeWhile} \, (\text{Not} \circ \text{is-volatile-Write}_{sb}) \, \text{xs}))

= \{\}

\text{by} \quad (\text{fastforce simp add: Read}_{sb} \, \text{aargh})

\text{show} \quad \text{?thesis}

\text{proof} \quad (\text{cases volatile})

\text{case True}

\text{from Cons.hyps} \quad [\text{OF no-inter}^\dagger \, \text{consis}^\dagger \, \text{RO}^\dagger]

\text{show} \quad \text{?thesis}

\text{by} \quad (\text{simp add: Read}_{sb} \, \text{True})

\text{next}

\text{case False}

250
note non-volatile=this

from Cons.hyps [OF no-inter’ consis’ RO’]
have hyp: reads-consistent False $O$ (flush sb m) xs.

show ?thesis
proof (cases a ∈ $O$)
case False
with hyp show ?thesis
by (simp add: Read$_{ab}$ non-volatile False)
next
case True
from no-inter True have a-notin: a ∉ outstanding-refs is-non-volatile-Write$_{sb}$ sb
by blast

with no-volatile-Write$_{sb}$-sb
have a ∉ outstanding-refs is-Write$_{sb}$ sb
apply (clarsimp simp add: outstanding-refs-conv is-Write$_{sb}$-def split: memref.splits)
apply force
done

from flush-unchanged-addresses [OF this] hyp val

show ?thesis
by (simp add: Read$_{sb}$ non-volatile True)
qed
qed

next
case Prog$_{sb}$ with Cons show ?thesis
by auto
next
case (Ghost$_{sb}$ A L R W)
from consis obtain consis’: reads-consistent False ($O$ ∪ $A$ − $R$) m xs
by (simp add: Ghost$_{sb}$)

from RO
have RO’: read-only-reads ($O$ ∪ $A$ − $R$) (takeWhile (Not ◦ is-volatile-Write$_{sb}$) xs) ⊆ RO
by (auto simp add: Ghost$_{sb}$)

from no-inter
have no-inter’: outstanding-refs is-non-volatile-Write$_{sb}$ sb ∩
(outstanding-refs is-non-volatile-Write$_{sb}$ xs ∪
outstanding-refs is-non-volatile-Write’$_{sb}$ xs ∪
(outstanding-refs is-non-volatile-Read$_{sb}$ (dropWhile (Not ◦ is-volatile-Write$_{sb}$) xs)
∪
(outstanding-refs is-non-volatile-Read$_{sb}$ (takeWhile (Not ◦ is-volatile-Write$_{sb}$) xs)
− RO) ∪
($O$ ∪ $A$ − $R$ ∪ all-acquired (takeWhile (Not ◦ is-volatile-Write$_{sb}$) xs)))

251
\( \{\} \) by (fastforce simp add: Ghostsb aargh)

from Cons.hyps [OF no-inter' consis' RO']
show ?thesis
by (clarsimp simp add: Ghostsb)
qed

lemma reads-consistent-flush-all-until-volatile-write-aux':
assumes no-reads: outstanding-refs is-volatile-Readsb xs = \{\}
assumes read-only-reads-RO: read-only-reads \( O' \) (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( x_s \)) \( x_s \) \( \subseteq \) RO
shows \( \bigwedge m. \ [\text{reads-consistent } O' \text{ } m \text{ } x_s; \ \forall i < \text{ length } t_s.\) let (\( p_t, is_t, \varnothing_t, D_t, O_t, R_t \)) = tsi in
  outstanding-refs is-non-volatile-Write\( s_b \) (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( s_b \)) \( \cap \) (outstanding-refs is-volatile-Write\( s_b \) \( x_s \) \( \cup \) outstanding-refs is-non-volatile-Write\( s_b \) \( x_s \) \( \cup \) outstanding-refs is-non-volatile-Read\( s_b \) (dropWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( x_s \)) \( \cup \) (outstanding-refs is-non-volatile-Read\( s_b \) (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( s_b \)) \( - \) RO) \( \cup \) (\( O' \cup \) all-acquired (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( x_s \)))
\) = \{\}
\] \Rightarrow \text{reads-consistent } O' \text{ } (\text{flush-all-until-volatile-write } t_s \text{ } m) \text{ } x_s

proof (induct ts)
case Nil thus ?case by simp
next
case (Cons t ts)
have consis: reads-consistent False \( O' \text{ } m \text{ } x_s \) by fact

obtain \( p_t, is_t, \varnothing_t, D_t, \varnothing_t, sb_t \)
  where t: \( t = (p_t, is_t, \varnothing_t, sb_t, D_t, O_t, R_t) \)
  by (cases t)
obtain no-inter: outstanding-refs is-non-volatile-Write\( s_b \) (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( s_b_t \) \( \cap \) (outstanding-refs is-volatile-Write\( s_b \) \( x_s \) \( \cup \) outstanding-refs is-non-volatile-Write\( s_b \) \( x_s \) \( \cup \) outstanding-refs is-non-volatile-Read\( s_b \) (dropWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( x_s \)) \( \cup \) (outstanding-refs is-non-volatile-Read\( s_b \) (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( s_b \)) \( - \) RO) \( \cup \) (\( O' \cup \) all-acquired (takeWhile (Not \( \circ \) is-volatile-Write\( s_b \)) \( x_s \)))))
\)
= {} and
no-inter': \forall i < \text{length } ts.
let (p, is, \emptyset, sb, D, O) = tl in
outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not \circ \text{is-volatile-Write}_{sb} sb) \cap
(outstanding-refs is-volatile-Write_{sb} xs \cup
(outstanding-refs is-non-volatile-Write_{sb} xs \cup
outstanding-refs is-non-volatile-Read_{sb} (dropWhile (Not \circ \text{is-volatile-Write}_{sb} xs)

\cup
(outstanding-refs is-non-volatile-Read_{sb} (takeWhile (Not \circ \text{is-volatile-Write}_{sb} xs))

\cup
(\emptyset \cup \text{all-acquired} (\text{takeWhile} (Not \circ \text{is-volatile-Write}_{sb} xs)))

= {} 

\text{proof --}
\begin{itemize}
  \item \text{show } ?\text{thesis}
  \item \text{using } Cons.prems (2) [\text{rule-format, of } 0]
  \item \text{apply } (\text{clarsimp simp add: t})
  \item \text{apply } clarsimp
  \item \text{using } Cons.prems (2)
  \item \text{apply --}
  \item \text{subgoal for } i
  \item \text{apply } (\text{drule-tac } x=\text{Suc } i \text{ in spec})
  \item \text{apply } (\text{clarsimp simp add: Let-def simp del: o-apply})
  \item \text{done}
  \item \text{done}
\end{itemize}
\text{qed}

\text{have } out1: \text{outstanding-refs is-volatile-Write}_{sb}
\begin{itemize}
  \item (\text{takeWhile} (Not \circ \text{is-volatile-Write}_{sb} sb_{i}) = \{} 
\end{itemize}
\text{by } (\text{auto simp add: outstanding-refs-conv dest: set-takeWhileD})

\text{from } \text{reads-consistent-flush-other'} \begin{itemize}
  \item \text{OF } out1 \text{ no-inter consis read-only-reads-RO }
\end{itemize}
\text{have } \text{reads-consistent } \text{False } O' \begin{itemize}
  \item \text{(flush } (\text{takeWhile} (Not \circ \text{is-volatile-Write}_{sb} sb_{i}) m) xs.
\end{itemize}
\text{from } \text{Cons.hyps } \begin{itemize}
  \item \text{OF } this \text{ no-inter '}
\end{itemize}
\text{show } ?\text{case}
\begin{itemize}
  \item \text{by } (\text{simp add: t})
\end{itemize}
\text{qed}

\text{lemma } \text{in-outstanding-refs-cases } \begin{itemize}
  \item a \in \text{outstanding-refs P xs } \Rightarrow
\begin{itemize}
  \item (\forall \text{volatile sop v A L R W. } \text{(Write}_{sb} \text{volatile a sop v A L R W)} \in \text{set xs } \Rightarrow \text{P}
  \item (\text{Write}_{sb} \text{volatile a sop v A L R W}) \Rightarrow \text{C})
\end{itemize}
\end{itemize}
(\forall \text{volatile } t \ v. \ \text{Read}_{ab} \text{volatile } a \ t \ v) \in \text{set } xs \implies P (\text{Read}_{ab} \text{volatile } a \ t \ v) \implies C
\implies C
\text{apply (clarsimp simp add: outstanding-refs-conv)}
\text{subgoal for } x
\text{apply (case-tac } x\text{)}
\text{apply fastforce}\+
\text{done}
\text{done}

\text{lemma dropWhile-Cons: } (\text{dropWhile } P \ xs) = x#ys \implies \neg P x
\text{apply (induct xs)}
\text{apply (auto split: if-split-asm)}
\text{done}

\text{lemma reads-consistent-dropWhile:}
\text{reads-consistent pending-write } O m (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) \ sb) = \text{reads-consistent True } O m (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) \ sb)
\text{apply (case-tac } (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) \ sb)\text{)}
\text{apply (simp only:)}
\text{apply simp}
\text{apply (frule dropWhile-Cons)}
\text{apply (auto split: memref.splits)}
\text{done}

\text{theorem reads-consistent-flush-all-until-volatile-write:}
\text{\forall i m pending-write. } [\text{valid-ownership-and-sharing } S \ ts; i < \text{length } ts; ts!i = (p, is, \theta, sb, D, O, R); \text{reads-consistent pending-write } O m sb ]
\implies \text{reads-consistent True } (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) \ sb) \ O) \ (\text{flush-all-until-volatile-write } ts m) (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) \ sb)
\text{proof (induct ts)}
\text{case Nil thus } ?\text{case by simp}
\text{next}
\text{case } (\text{Cons } t \ ts)
\text{note } i\text{-bound } = \langle i < \text{length } (t \# ts)\rangle
\text{note } ts\text{-}i = \langle (t \# ts) \oplus i = (p, is, \theta, sb, D, O, R)\rangle
\text{note } consis = \langle \text{reads-consistent pending-write } O m sb \rangle
\text{note } valid = \langle \text{valid-ownership-and-sharing } S (t\#ts) \rangle
\text{then interpret } \text{valid-ownership-and-sharing } S t\#ts.
\text{from } \text{valid-ownership-and-sharing-tl } [\text{OF valid} \text{ have valid'} \text{ valid'}: \text{valid-ownership-and-sharing } S \ ts\rangle
\text{obtain } p_t \ is_t \ O_t \ R_t \ D_t \ \theta_t \ sb_t
\text{ where } t \text{: } t=(p_t, is_t, \theta_t, sb_t, D_t, O_t, R_t)
\text{by (cases } t\text{)}
\text{show } ?\text{case}
\text{proof (cases } i\text{)}
\text{case } 0
with ts·i t have sb-eq: sb=sb·i
  by simp

let ?take-sb = (takeWhile (Not ◦ is-volatile-Write sb) sb)
let ?drop-sb = (dropWhile (Not ◦ is-volatile-Write sb) sb)

from reads-consistent-append [of pending-write O m ?take-sb ?drop-sb] consis
  have consis': reads-consistent True (acquired True ?take-sb O) (flush ?take-sb m)
  ?drop-sb
  apply (cases outstanding-refs is-volatile-Write sb (takeWhile (Not ◦ is-volatile-Write sb)
  sb) ≠ { })
  apply clarsimp
  apply clarsimp
  apply (simp add: reads-consistent-dropWhile [of pending-write])
  done

from reads-consistent-drop-volatile-writes-no-volatile-reads [OF consis]
  have no-vol-Read sb: outstanding-refs is-volatile-Read sb (dropWhile (Not ◦ is-volatile-Write sb)
  sb) = { }.
  hence outstanding-refs (Not ◦ is-volatile-Read sb) (dropWhile (Not ◦ is-volatile-Write sb)
  sb)

  = outstanding-refs (λs. True) (dropWhile (Not ◦ is-volatile-Write sb) sb)
  by (auto simp add: outstanding-refs-conv)

  have ∀i<length ts.
    let (p, is, θ, sb, sb', D, O, R) = ts ! i
    in outstanding-refs (Not ◦ is-volatile-Read sb) (dropWhile (Not ◦ is-volatile-Write sb)
    sb) ∩ outstanding-refs is-non-volatile-Write sb (takeWhile (Not ◦ is-volatile-Write sb)
    sb')
  = { }
  proof –
  { }
  fix j p·j is·j O·j R·j D·j θ·j sb·j x
  assume j-bound: j < length ts
  assume ts·j: ts ! j = (p·j, is·j, θ·j, D·j, O·j, R·j)
  assume x-in-sb: x ∈ outstanding-refs (Not ◦ is-volatile-Read sb) (dropWhile (Not ◦
  is-volatile-Write sb) sb)
  assume x-in-j: x ∈ outstanding-refs is-non-volatile-Write sb (takeWhile (Not ◦
  is-volatile-Write sb) sb)
  have False
  proof –
  from outstanding-non-volatile-write-not-volatile-read-disj [rule-format, of Suc j 0,
  simplified, OF j-bound ts·j t]
  sb-eq x-in-sb x-in-j
  show ?thesis
  by auto
  qed

  }
thus ?thesis
by (auto simp add: Let-def)
qed

from reads-consistent-flush-all-untill-volatile-write-aux [OF no-vol-Read sb consis' this]
show ?thesis
by (simp add: t sb-eq del: o-apply)

next

  case (Suc k)

  with i-bound have k-bound: k < length ts
  by auto

  from ts-i Suc have ts-k: ts ! k = (p, is, δ, sb, D, O, R)
  by simp

  have reads-consistent False O (flush (takeWhile (Not ◦ is-volatile-Write sb) sb t) m) sb
  proof
    have no-vW: outstanding-refs is-volatile-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb t) = {}
    apply (clarsimp simp add: outstanding-refs-conv )
    apply simp
    done

  from consis have consis': reads-consistent False O m sb
  by (cases pending-write) (auto intro: reads-consistent-pending-write-antimono)

  note disj = outstanding-non-volatile-write-disj [where i=0, OF - i-bound [simplified Suc], simplified, OF t ts-k ]

  from reads-consistent-flush-other' [OF no-vW disj consis' subset-refl]
  show ?thesis .
  qed
  from Cons.hyps [OF valid' k-bound ts-k this]
  show ?thesis
  by (simp add: t)
  qed
  qed

lemma split-volatile-Write sb-in-outstanding-refs:
a ∈ outstanding-refs is-volatile-Write sb xs ⇔ (∃ sop v ys zs A L R W. xs = ys@(Write sb True a sop v A L R W #zs))
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  have a-in: a ∈ outstanding-refs is-volatile-Write sb (x # xs) by fact
show ?case
proof (cases x)
  case (Write_{sb} volatile a' sop v A L R W)
  show ?thesis
  proof (cases volatile)
    case False
    from a-in have a ∈ outstanding-refs is-volatile-Write_{sb} xs
    by (auto simp add: False Write_{sb})
    from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W'' ys zs
    where xs=ys@Write_{sb} True a sop'' v'' A'' L'' R'' W''#zs
    by auto
    hence x#xs = (x#ys)@Write_{sb} True a sop'' v'' A'' L'' R'' W''#zs
    by auto
    thus ?thesis
    by blast
  next
  case True
  note volatile = this
  show ?thesis
  proof (cases a'=a)
  case False
  with a-in have a ∈ outstanding-refs is-volatile-Write_{sb} xs
  by (auto simp add: volatile Write_{sb})
  from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W'' ys zs
  where xs=ys@Write_{sb} True a sop'' v'' A'' L'' R'' W''#zs
  by auto
  hence x#xs = (x#ys)@Write_{sb} True a sop'' v'' A'' L'' R'' W''#zs
  by auto
  thus ?thesis
  by blast
  next
  case Read_{sb}
  from a-in have a ∈ outstanding-refs is-volatile-Write_{sb} xs
  by (auto simp add: Read_{sb})
  from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W'' ys zs
  where xs=ys@Write_{sb} True a sop'' v'' A'' L'' R'' W''#zs
  by auto
  hence x#xs = (x#ys)@Write_{sb} True a sop'' v'' A'' L'' R'' W''#zs
  by auto
  thus ?thesis
  by blast
  next
case Prog$_{sb}$
from a-in have a ∈ outstanding-refs is-volatile-Write$_{sb}$ xs
  by (auto simp add: Prog$_{sb}$)
from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W'' y y s zs
  where xs=ys@Write$_{sb}$ True a sop'' v'' A'' L'' R'' W''#zs
  by auto
hence x#xs = (x#ys)@Write$_{sb}$ True a sop'' v'' A'' L'' R'' W''#zs
  by auto
thus ?thesis
  by auto
qed

case Ghost$_{sb}$
from a-in have a ∈ outstanding-refs is-volatile-Write$_{sb}$ xs
  by (auto simp add: Ghost$_{sb}$)
from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W'' y y s zs
  where xs=ys@Write$_{sb}$ True a sop'' v'' A'' L'' R'' W''#zs
  by auto
hence x#xs = (x#ys)@Write$_{sb}$ True a sop'' v'' A'' L'' R'' W''#zs
  by auto
thus ?thesis
  by auto
qed

lemma sharing-consistent-mono-shared:
\(\forall S \subseteq S'. \quad S \subseteq \text{dom } S' \Rightarrow \text{sharing-consistent } S \circ \text{sb} \Rightarrow \text{sharing-consistent } S' \circ \text{sb}\)
apply (induct sb)
apply simp
subgoal for a sb S S' O
apply (case-tac a)
apply clarsimp
  subgoal for volatile a D f v A L R W
  apply (frule-tac A=S and B=S' and C=R and x=W in augment-mono-aux)
  apply (frule-tac A=S ⊕ W R and B=S' ⊕ W R and C=L in restrict-mono-aux)
  apply blast
  done
apply clarsimp
apply clarsimp
apply clarsimp
subgoal for A L R W
apply (frule-tac A=S and B=S' and C=R and x=W in augment-mono-aux)
apply (frule-tac A=S ⊕ W R and B=S' ⊕ W R and C=L in restrict-mono-aux)
apply blast
done
done
done

lemma sharing-consistent-mono-owns:
\(\forall O \subseteq O'. \quad S.\)

258
\(O \subseteq O' \implies \text{sharing-consistent } S \ O \ sb \implies \text{sharing-consistent } S \ O' \ sb\)

apply (induct sb)
apply simp

subgoal for a sb O O' S
apply (case-tac a)
apply clarsimp

subgoal for volatile a D f v A L R W
apply (frule-tac A=O and B=O' and C=A in union-mono-aux)
apply (frule-tac A=O ∪ A and B=O' ∪ A and C=R in set-minus-mono-aux)
apply fastforce
done
apply clarsimp
apply clarsimp
apply clarsimp
subgoal for A L R W
apply (frule-tac A=O and B=O' and C=A in union-mono-aux)
apply (frule-tac A=O ∪ A and B=O' ∪ A and C=R in set-minus-mono-aux)
apply fastforce
done
done
done
done
done
done
done
done

done

done

done

primrec all-shared :: 'a memref list ⇒ addr set
where
  all-shared [] = {}
| all-shared (i#is) =
    (case i of
        Write sb volatile - - - A L R W ⇒ (if volatile then R ∪ all-shared is else all-shared is)
    | Ghost sb A L R W ⇒ R ∪ all-shared is
    | - ⇒ all-shared is)

lemma sharing-consistent-all-shared:
\(\forall S \ O. \ \text{sharing-consistent } S \ O \ sb \implies \text{all-shared } sb \subseteq \text{dom } S \cup O\)
apply (induct sb)
apply clarsimp
subgoal for a
apply (case-tac a)
apply (fastforce split: memref.splits if-split-asm)
apply clarsimp
apply clarsimp
apply clarsimp
apply fastforce
done
done
done

lemma sharing-consistent-share-all-shared:
\(\forall S. \ \text{dom } (\text{share } sb \ S) \subseteq \text{dom } S \cup \text{all-shared } sb\)
proof (induct sb)
case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
proof (cases x)
  case (Write$_{sb}$ volatile a sop t A L R W)
  show ?thesis
proof (cases volatile)
    case True
    from Cons.hyps [of ($\mathcal{S} \oplus_W R \ominus_A L$)]
    show ?thesis
    by (auto simp add: Write$_{sb}$ True)
next
  case False with Cons Write$_{sb}$ show ?thesis by auto
qed
next
  case Read$_{sb}$ with Cons show ?thesis by auto
next
  case Prog$_{sb}$ with Cons show ?thesis by auto
next
  case (Ghost$_{sb}$ A L R W)
  from Cons.hyps [of ($\mathcal{S} \oplus_W R \ominus_A L$)]
  show ?thesis
  by (auto simp add: Ghost$_{sb}$)
qed

primrec all-unshared :: 'a memref list ⇒ addr set
where
  all-unshared [] = {}
| all-unshared (i#is) =
    (case i of
      Write$_{sb}$ volatile - - - A L R W ⇒ (if volatile then L ∪ all-unshared is else all-unshared is)
    | Ghost$_{sb}$ A L R W ⇒ L ∪ all-unshared is
    | - ⇒ all-unshared is)

lemma all-unshared-append: all-unshared (xs @ ys) = all-unshared xs ∪ all-unshared ys
apply (induct xs)
apply simp
subgoal for a
apply (case-tac a)
apply auto
done
done

lemma freshly-shared-owned:
∀S O. sharing-consistent S O sb ⇒ dom (share sb S) − dom S ⊆ O

proof (induct sb)
  case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case False
  with Cons Write sb show ?thesis by auto
next
case True
from Cons.hyps [where S = (S ⊕ W R ⊕ A L) and O = (O ∪ A − R)] Cons.prems
show ?thesis by (auto simp add: Write sb True)
qed
next
case Read sb with Cons show ?thesis by auto
next
case Prog sb with Cons show ?thesis by auto
next
case (Ghost sb A L R W)
  with Cons.hyps [where S = (S ⊕ W R ⊕ A L) and O = (O ∪ A − R)] Cons.prems
  show ?thesis by auto
qed
qed

lemma unshared-all-unshared:
∀S O. sharing-consistent S O sb ⇒ dom S − dom (share sb S) ⊆ all-unshared sb

proof (induct sb)
  case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case False
  with Cons Write sb show ?thesis by auto
next
case True
from Cons.hyps [where S = (S ⊕ W R ⊕ A L) and O = (O ∪ A − R)] Cons.prems
show ?thesis by (auto simp add: Write sb True)
qed
next
case Read sb with Cons show ?thesis by auto
next
  case Prog\sb with Cons show ?thesis by auto
next
  case (Ghost\sb \ A \ L \ R \ W)
      with Cons.hyps [where \s = (\s \ominus \ W \ominus A \ L) \ and \ O = (O \cup A - R)] Cons.prems show ?thesis by auto
    qed
  qed
lemma unshared-acquired-or-owned:
  \A \s \ O. sharing-consistent \s \ O \sb \implies \all-unshared \sb \subseteq \all-acquired \sb \cup O
apply (induct \sb)
apply simp
subgoal for a
apply (case-tac a)
apply auto+
done
done
lemma all-shared-acquired-or-owned:
  \A \s \ O. sharing-consistent \s \ O \sb \implies \all-shared \sb \subseteq \all-acquired \sb \cup O
apply (induct \sb)
apply simp
subgoal for a
apply (case-tac a)
apply auto+
done
done
lemma sharing-consistent-preservation:
  \A \s \ O. [sharing-consistent \s \ O \sb; \all-acquired \sb \cap \dom \s - \dom \s' = \{\}; \all-unshared \sb \cap \dom \s' - \dom \s = \{\}] \implies sharing-consistent \s' \ O \sb
proof (induct \sb)
  case Nil thus ?case by simp
next
  case (Cons \x \sb)
  have consis: sharing-consistent \s \ O (\x \# \sb) by fact
  have removed-cond: all-acquired (\x \# \sb) \cap \dom \s - \dom \s' = \{\} by fact
  have new-cond: all-unshared (\x \# \sb) \cap \dom \s' - \dom \s = \{\} by fact
  show ?case
  proof (cases \x)
    case (Write\sb volatile \a sop \ v \ A \ L \ R \ W)
    show ?thesis
      proof (cases volatile)
        case False with Write\sb Cons show ?thesis by auto
      qed
    qed
  qed

262
next
  case True
  from consis obtain
  A: A ⊆ dom S ∪ O and
  L: L ⊆ A and
  A-R: A ∩ R = {} and
  R: R ⊆ O and
  consis': sharing-consistent (S ⊕ W R ⊖ A L) (O ∪ A − R) sb
  by (clarsimp simp add: Write_{sb} True)

  from removed-cond obtain rem-cond: (A ∪ all-acquired sb) ∩ dom S ⊆ dom S' by
  (clarsimp simp add: Write_{sb} True)
  hence rem-cond': all-acquired sb ∩ dom (S ⊕ W R ⊖ A L) − dom (S' ⊕ W R ⊖ A L) =
  {} by auto

  from new-cond obtain (L ∪ all-unshared sb) ∩ dom S' ⊆ dom S by (clarsimp simp add: Write_{sb} True)
  hence new-cond': all-unshared sb ∩ dom (S' ⊕ W R ⊖ A L) − dom (S ⊕ W R ⊖ A L) =
  {} by auto

  from Cons.hyps [OF consis' rem-cond' new-cond']
  have sharing-consistent (S' ⊕ W R ⊖ A L) (O ∪ A − R) sb.
  moreover
  from A rem-cond have A ⊆ dom S' ∪ O
  by auto
  moreover note L A-R R
  ultimately show ?thesis
  by (auto simp add: Write_{sb} True)
  qed
next
  case (Ghost_{sb} A L R W)
  from consis obtain
  A: A ⊆ dom S ∪ O and
  L: L ⊆ A and
  A-R: A ∩ R = {} and
  R: R ⊆ O and
  consis': sharing-consistent (S ⊕ W R ⊖ A L) (O ∪ A − R) sb
  by (clarsimp simp add: Ghost_{sb})

  from removed-cond obtain rem-cond: (A ∪ all-acquired sb) ∩ dom S ⊆ dom S' by
  (clarsimp simp add: Ghost_{sb})
  hence rem-cond': all-acquired sb ∩ dom (S ⊕ W R ⊖ A L) − dom (S' ⊕ W R ⊖ A L) =
  {} by auto

  from new-cond obtain (L ∪ all-unshared sb) ∩ dom S' ⊆ dom S by (clarsimp simp add: Ghost_{sb})
hence new-cond': all-unshared sb ∩ dom (S' ⊕ W R ∋ A L) − dom (S ⊕ W R ∋ A L) = 
{ }
by auto

from Cons.hyps [OF cons' rem-cond' new-cond']
have sharing-consistent (S' ⊕ W R ∋ A L) (O ∪ A − R) sb.
moreover
from A rem-cond have A ⊆ dom S' ∪ O
by auto
moreover note L A-R R
ultimately show ?thesis
by (auto simp add: Ghost sb)
qed (insert Cons, auto)

qed

lemma (in sharing-consis) sharing-consis-preservation:
assumes dist:
∀ i < length ts. let (⋅,⋅,⋅,sb,⋅,⋅,⋅) = ts!i in
  all-acquired sb ∩ dom S − dom S' = {} ∧ all-unshared sb ∩ dom S' − dom S = 
{ }
shows sharing-consis S' ts
proof
  fix i p is O R D ∅ sb
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,∅,sb,D,O,R)
  show sharing-consistent S' O sb
proof −
  from sharing-consis [OF i-bound ts-i]
  have consis: sharing-consistent S O sb.
  from dist [rule-format, OF i-bound] ts-i
  obtain
    acq: all-acquired sb ∩ dom S − dom S' = {} and
    uns: all-unshared sb ∩ dom S' − dom S = {}
    by auto
  from sharing-consistent-preservation [OF consis acq uns]
  show ?thesis .
  qed

qed

lemma (in sharing-consis) sharing-consis-shared-exchange:
assumes dist:
∀ i < length ts. let (⋅,⋅,⋅,sb,⋅,⋅,⋅) = ts!i in
  ∀ a ∈ all-acquired sb. S' a = S a
shows sharing-consis S' ts
proof
  fix i p is O R D ∅ sb
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,∅,sb,D,O,R)
  show sharing-consistent S' O sb
proof −
from sharing-consis [OF i-bound ts-i]

have consis: sharing-consistent $S \ O$ sb.

from dist [rule-format, OF i-bound] ts-i

obtain
dist-sb: $\forall a \in$ all-acquired sb. $S' a = S a$

by auto

from sharing-consistent-shared-exchange [OF dist-sb consis]

show $\exists$thesis.

qed

qed

lemma all-acquired-takeWhile: all-acquired (takeWhile P sb) $\subseteq$ all-acquired sb

proof –

from all-acquired-append [of takeWhile P sb dropWhile P sb]

show $\exists$thesis

by auto

qed

lemma all-acquired-dropWhile: all-acquired (dropWhile P sb) $\subseteq$ all-acquired sb

proof –

from all-acquired-append [of takeWhile P sb dropWhile P sb]

show $\exists$thesis

by auto

qed

lemma acquired-share-owns-shared:

assumes consis: sharing-consistent $S \ O$ sb

shows acquired pending-write sb $O \cup$ dom (share sb $S$) $\subseteq$ $O \cup$ dom $S$

proof –

from acquired-all-acquired have acquired pending-write sb $O \subseteq$ $O \cup$ all-acquired sb.

moreover

from sharing-consistent-all-acquired [OF consis] have all-acquired sb $\subseteq$ dom $S \cup O$.

moreover

from sharing-consistent-share-all-shared have dom (share sb $S$) $\subseteq$ dom $S \cup$ all-shared sb.

moreover

from sharing-consistent-all-shared [OF consis] have all-shared sb $\subseteq$ dom $S \cup O$.

ultimately

show $\exists$thesis

by blast

qed

lemma acquired-owns-shared:

assumes consis: sharing-consistent $S \ O$ sb

shows acquired True sb $O \subseteq$ $O \cup$ dom $S$

using acquired-share-owns-shared [OF consis]

by blast

265
lemma share-owns-shared:
  assumes consis: sharing-consistent $S \ O \ sb$
  shows dom (share sb $S$) $\subseteq O \cup \text{dom } S$
using acquired-share-owns-shared [OF consis]
by blast

lemma all-shared-append: all-shared (xs@ys) = all-shared xs $\cup$ all-shared ys
by (induct xs) (auto split: memref.splits)

lemma acquired-union-notin-first:
  $\forall$ pending-write A B. a $\notin$ acquired pending-write sb (A $\cup$ B) $\implies$ a $\notin$ A $\implies$ a $\in$ acquired pending-write sb B
proof (induct sb)
case Nil thus ?case by (auto split: if-split-asm)
next
case (Cons x sb)
then obtain a-notin-A: a $\notin$ A and
  a-acq: a $\in$ acquired pending-write (x # sb) (A $\cup$ B)
by blast
show ?case
proof (cases x)
case (Write sb volatile a' sop v A' L R W)
  show ?thesis
  proof (cases volatile)
    case False
    with Write sb Cons show ?thesis by simp
  next
    case True
    from a-acq have a-acq': a $\in$ acquired True sb (A $\cup$ B $\cup$ A' $\setminus$ R)
    by (simp add: Write sb volatile True)
  have (A $\cup$ B $\cup$ A' $\setminus$ R) $\subseteq$ (A $\cup$ (B $\cup$ A' $\setminus$ R))
  by auto
from acquired-mono-in [OF a-acq' this]
have a $\in$ acquired True sb (A $\cup$ (B $\cup$ A' $\setminus$ R)).
from Cons.hyps [OF this a-notin-A]
  have a $\in$ acquired True sb (B $\cup$ A' $\setminus$ R).
then
show ?thesis by (simp add: Write sb volatile True)
next
case False
from a-acq have a-acq': a $\in$ acquired True sb (A' $\setminus$ R)
  by (simp add: Write sb volatile False)
then
show ?thesis
by (simp add: Write sb volatile False)
qed
qed
next

case (Ghostsb A’ L R W)
show ?thesis
proof (cases pending-write)
  case True
  from a-acq have a-acq’: a ∈ acquired True sb (A ∪ B ∪ A’ − R)
  by (simp add: Ghostsb True)
  have (A ∪ B ∪ A’ − R) ⊆ (A ∪ (B ∪ A’ − R))
  by auto
  from acquired-mono-in [OF a-acq’ this]
  have a ∈ acquired True sb (A ∪ (B ∪ A’ − R)).
  from Cons.hyps [OF this a-notin-A]

  have a ∈ acquired True sb (B ∪ A’ − R).
  then
  show ?thesis by (simp add: Ghostsb True)
next
  case False
  from a-acq have a-acq’: a ∈ acquired False sb (A ∪ B)
  by (simp add: Ghostsb False)
  from Cons.hyps [OF this a-notin-A]
  show ?thesis
  by (simp add: Ghostsb False)
qed
qed

lemma split-all-acquired-in:
a ∈ all-acquired xs ⇒
(∃ sop a’ v ys zs A L R W. xs = ys @ Write sb True a’ sop v A L R W# zs ∧ a ∈ A) ∨
(∃ A L R W ys zs. xs = ys @ Ghostsb A L R W# zs ∧ a ∈ A)
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
have a-in: a ∈ all-acquired (x # xs) by fact
show ?case
proof (cases x)
case (Write sb volatile a’ sop v A L R W)
show ?thesis
proof (cases volatile)
case False
from a-in have \( a \in \text{all-acquired} \ \text{xs} \)
by (auto simp add: False Write
from Cons.hyps [OF this]
have \( \exists \text{sop} \ a' \ v \ ys \ zs \ A \ L \ R \ W. \ \text{xs} = \text{ys} @ \text{Write}_{sb} \ \text{True} \ a' \ \text{sop} \ v \ A \ L \ R \ W \# \ zs \ \land \ a \in A \) \lor
(\exists A \ L \ R \ W \ ys \ zs. \ \text{xs} = \text{ys} @ \text{Ghost}_{sb} \ A \ L \ R \ W \# \ zs \ \land \ a \in A \) (is \ ?\text{write} \lor \ ?\text{ghst}).
then
show \ ?\text{thesis}
proof
assume \ ?\text{write}
then
obtain \text{sop}'' \ a'' \ v'' \ A'' \ L'' \ R'' \ W'' \ ys \ zs
where \text{xs} = \text{ys} @ \text{Write}_{sb} \ \text{True} \ a'' \ \text{sop}'' \ v'' \ A'' \ L'' \ R'' \ W'' \# \ zs \ \text{and} \ a\text{-in}: \ a \in A''
by auto
hence \text{x#xs} = (\text{x#ys}) @ \text{Write}_{sb} \ \text{True} \ a'' \ \text{sop}'' \ v'' \ A'' \ L'' \ R'' \ W'' \# \ zs
by auto
thus \ ?\text{thesis}
using a-in
by blast
next
assume \ ?\text{ghst}
then obtain A'' L'' R'' W'' \ ys \ zs \ where
\text{xs} = \text{ys} @ \text{Ghost}_{sb} \ A'' \ L'' \ R'' \ W'' \# \ zs \ \text{and} \ a\text{-in}: \ a \in A''
by auto
hence \text{x#xs} = (\text{x#ys}) @ \text{Ghost}_{sb} \ A'' \ L'' \ R'' \ W'' \# \ zs
by auto
thus \ ?\text{thesis}
using a-in
by blast
qed
next
case True
note volatile = this
show \ ?\text{thesis}
proof (cases a \in A)
case False
with a-in have \( a \in \text{all-acquired} \ \text{xs} \)
by (auto simp add: volatile Write_{sb})
from Cons.hyps [OF this]
have \( \exists \text{sop} \ a' \ v \ ys \ zs \ A \ L \ R \ W. \ \text{xs} = \text{ys} @ \text{Write}_{sb} \ \text{True} \ a' \ \text{sop} \ v \ A \ L \ R \ W \# \ zs \ \land \ a \in A \) \lor
(\exists A \ L \ R \ W \ ys \ zs. \ \text{xs} = \text{ys} @ \text{Ghost}_{sb} \ A \ L \ R \ W \# \ zs \ \land \ a \in A \) (is \ ?\text{write} \lor \ ?\text{ghst}).
then
show \ ?\text{thesis}
proof
assume \ ?\text{write}
then
obtain \text{sop}'' \ a'' \ v'' \ A'' \ L'' \ R'' \ W'' \ ys \ zs
where \(xs = ys @ \text{Write}_{ab} \text{ True } a'' \text{sop}'' v'' A'' L'' R'' W'' \#zs\) and \(a \in A''\) by auto

hence \(x \# xs = (x \# ys) @ \text{Write}_{ab} \text{ True } a'' \text{sop}'' v'' A'' L'' R'' W'' \#zs\)

by auto

thus \(?\text{thesis}\)

using \(a\text{-in}\)

by blast

next

assume \(?\text{ghost}\)

then obtain \(A'' L'' R'' W'' ys zs\) where

\(xs = ys @ \text{Ghost}_{ab} A'' L'' R'' W'' \#zs\) and \(a \in A''\)

by auto

hence \(x \# xs = (x \# ys) @ \text{Ghost}_{ab} A'' L'' R'' W'' \#zs\)

by auto

thus \(?\text{thesis}\)

using \(a\text{-in}\)

by blast

qed

next

case True

then have \(x \# xs = [] @(\text{Write}_{ab} \text{ True } a' \text{sop} v A L R W \#xs)\)

by \(\text{(simp add: Write}_{ab} \text{ volatile True)}\)

thus \(?\text{thesis}\)

using \(\text{True}\)

by blast

qed

qed

next

case \(\text{Read}_{ab}\)

from \(a\text{-in}\) have \(a \in \text{all-acquired } xs\)

by \(\text{(auto simp add: Read}_{ab})\)

from \(\text{Cons.hyps [OF this]}\)

have \((\exists \text{sop } a' v ys zs A L R W. \ xs = ys @ \text{Write}_{ab} \text{ True } a' \text{sop} v A L R W \# zs \land a \in A) \lor\)

\((\exists A L R W ys zs. \ xs = ys @ \text{Ghost}_{ab} A L R W \# zs \land a \in A) \ (\text{is } ?\text{write} \lor ?\text{ghost}).\)

then

show \(?\text{thesis}\)

proof

assume \(?\text{write}\)

obtain sop'' a'' v'' A'' L'' R'' W'' ys zs

where \(xs = ys @ \text{Write}_{ab} \text{ True } a'' \text{sop}'' v'' A'' L'' R'' W'' \#zs\) and \(a \in A''\)

by auto

hence \(x \# xs = (x \# ys) @ \text{Write}_{ab} \text{ True } a'' \text{sop}'' v'' A'' L'' R'' W'' \#zs\)

by auto

thus \(?\text{thesis}\)

using \(a\text{-in}\)

by blast

next

assume \(?\text{ghost}\)
then obtain $A'' L'' R'' W'' y s z s$ where
\[ x s = y s @ \text{Ghost}_{sb} A'' L'' R'' W'' # z s \] and \( a \in A'' \)
by auto
\[ \text{hence } x \# x s = (x \# y s) @ \text{Ghost}_{sb} A'' L'' R'' W'' # z s \]
by auto
\[ \text{thus } ? \text{thesis} \]
using a-in
by blast
qed
next
case Prog_{sb}
from a-in have \( a \in \text{all-acquired } x s \)
by (auto simp add: Prog_{sb})
from Cons.hyps [OF this]
have \((\exists \text{sop } a' v y s A L R W. x s = y s @ \text{Write}_{sb} \text{ True } a' \text{sop } v A L R W \# z s \land a \in A) \lor \)
\[ (\exists A L R W y s z s. x s = y s @ \text{Ghost}_{sb} A L R W \# z s \land a \in A) \) (is ?write \lor ?\text{ghst}).
then
show ?thesis
proof
assume ?write
then
obtain sop'' a'' v'' A'' L'' R'' W'' y s z s
where \( x s = y s @ \text{Write}_{sb} \text{ True } a'' \text{sop'' } v'' A'' L'' R'' W'' # z s \) and \( a \in A'' \)
by auto
\[ \text{hence } x \# x s = (x \# y s) @ \text{Write}_{sb} \text{ True } a'' \text{sop'' } v'' A'' L'' R'' W'' # z s \]
by auto
\[ \text{thus } ? \text{thesis} \]
using a-in
by blast
next
assume ?\text{ghst}
then obtain $A'' L'' R'' W'' y s z s$ where
\[ x s = y s @ \text{Ghost}_{sb} A'' L'' R'' W'' # z s \] and \( a \in A'' \)
by auto
\[ \text{hence } x \# x s = (x \# y s) @ \text{Ghost}_{sb} A'' L'' R'' W'' # z s \]
by auto
\[ \text{thus } ? \text{thesis} \]
using a-in
by blast
qed
next
case (\text{Ghost}_{sb} A L R W)
show ?thesis
proof (cases \( a \in A \))
\[ \text{case } \text{False} \]
with a-in have \( a \in \text{all-acquired } x s \)
by (auto simp add: Ghost_{sb})
from Cons.hyps [OF this]
have (∃ sop a' v ys zs A L R W. xs = ys @ Write sb True a' sop v A L R W # zs ∧ a ∈ A) ∨
(∃ A L R W ys zs. xs = ys @ Ghost sb A L R W# zs ∧ a ∈ A) (is ?write ∨ ?ghst).

then
show ?thesis
proof
assume ?write
then
obtain sop'' a'' v'' A'' L'' R'' W'' ys zs
where xs=ys@Write sb True a'' sop'' v'' A'' L'' R'' W''#zs and a-in: a ∈ A''
by auto
hence x#xs = (x#ys)@Write sb True a'' sop'' v'' A'' L'' R'' W''#zs
by auto
thus ?thesis
using a-in
by blast
next
assume ?ghst
then obtain A'' L'' R'' W'' ys zs where
xs=ys@Ghost sb A'' L'' R'' W''#zs and a-in: a ∈ A''
by auto
hence x#xs = (x#ys)@Ghost sb A'' L'' R'' W''#zs
by auto
thus ?thesis
using a-in
by blast
qed
next

then have x#xs=[][]@(Ghost sb A L R W#xs)
by (simp add: Ghost sb True)
thus ?thesis
using True
by blast
qed
qed

lemma split-Write sb-in-outstanding-refs:
a ∈ outstanding-refs is-Write sb xs ===> (∃ sop volatile v ys zs A L R W. xs = ys@(Write sb volatile a sop v A L R W#zs))
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
have a-in: a ∈ outstanding-refs is-Write sb (x # xs) by fact
show ?case
proof (cases x)
case (\texttt{Write}	extsubscript{sb} \texttt{volatile} \texttt{a}' \texttt{sop} \texttt{v A L R W})
\texttt{show} \ ?\texttt{thesis}
\texttt{proof} (\texttt{cases} \texttt{a'}=\texttt{a})
\texttt{case} \texttt{False}
\texttt{with} \texttt{a-in have} \texttt{a} \in \texttt{outstanding-refs} \texttt{is-Write}	extsubscript{sb} \texttt{xs}
\texttt{by} \ (\texttt{auto simp add: Write}	extsubscript{sb})
\texttt{from Cons.hyps [OF this] obtain} \texttt{sop'' volatile'' v'' A'' L'' R'' W'' ys zs}
\texttt{where} \texttt{xs=ys@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{hence} \texttt{x#xs} = (\texttt{x#ys}@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{thus} \ ?\texttt{thesis}
\texttt{by} \texttt{blast}
\texttt{next}
\texttt{case} \texttt{True}
\texttt{then have} \texttt{x#xs=}[]@(\texttt{Write}	extsubscript{sb} \texttt{volatile} \texttt{a sop} \texttt{v A L R W#xs})
\texttt{by} \ (\texttt{simp add: Write}	extsubscript{sb} \texttt{True})
\texttt{thus} \ ?\texttt{thesis}
\texttt{by} \texttt{blast}
\texttt{qed}
\texttt{next}
\texttt{case Read}	extsubscript{sb}
\texttt{from a-in have} \texttt{a} \in \texttt{outstanding-refs} \texttt{is-Write}	extsubscript{sb} \texttt{xs}
\texttt{by} \ (\texttt{auto simp add: Read}	extsubscript{sb})
\texttt{from Cons.hyps [OF this] obtain} \texttt{sop'' volatile'' v'' A'' L'' R'' W'' ys zs}
\texttt{where} \texttt{xs=ys@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{hence} \texttt{x#xs} = (\texttt{x#ys}@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{thus} \ ?\texttt{thesis}
\texttt{by} \texttt{blast}
\texttt{next}
\texttt{case Prog}	extsubscript{sb}
\texttt{from a-in have} \texttt{a} \in \texttt{outstanding-refs} \texttt{is-Write}	extsubscript{sb} \texttt{xs}
\texttt{by} \ (\texttt{auto simp add: Prog}	extsubscript{sb})
\texttt{from Cons.hyps [OF this] obtain} \texttt{sop'' volatile'' v'' A'' L'' R'' W'' ys zs}
\texttt{where} \texttt{xs=ys@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{hence} \texttt{x#xs} = (\texttt{x#ys}@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{thus} \ ?\texttt{thesis}
\texttt{by} \texttt{blast}
\texttt{next}
\texttt{case Ghost}	extsubscript{sb}
\texttt{from a-in have} \texttt{a} \in \texttt{outstanding-refs} \texttt{is-Write}	extsubscript{sb} \texttt{xs}
\texttt{by} \ (\texttt{auto simp add: Ghost}	extsubscript{sb})
\texttt{from Cons.hyps [OF this] obtain} \texttt{sop'' volatile'' v'' A'' L'' R'' W'' ys zs}
\texttt{where} \texttt{xs=ys@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{by} \texttt{auto}
\texttt{hence} \texttt{x#xs} = (\texttt{x#ys}@Write}	extsubscript{sb} \texttt{volatile'' a sop'' v'' A'' L'' R'' W''#zs}
\texttt{272}
by auto
thus ?thesis
by blast
qed
qed

lemma outstanding-refs-is-Write_{sb}-union:
  outstanding-refs is-Write_{sb} xs =
  (outstanding-refs is-volatile-Write_{sb} xs \cup outstanding-refs is-non-volatile-Write_{sb} xs)
apply (induct xs)
apply simp
subgoal for a
apply (case-tac a)
apply auto
done
done

lemma rtranclp-r-rtranclp: \([r^{**} x y; r y z]\) = \(r^{**} x z\)
  by auto

lemma r-rtranclp-rtranclp: \([r x y; r^{**} y z]\) = \(r^{**} x z\)
  by auto

lemma unshared-is-non-volatile-Write_{sb}: \(\forall S.
  [non-volatile-writes-unshared S sb; a \in \text{dom } S; a \not\in \text{all-unshared } sb] \implies
  a \not\in \text{outstanding-refs is-non-volatile-Write}_{sb} sb\)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write_{sb} volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case False
  with Cons Write_{sb} show ?thesis by auto
next
case True
  from Cons.hyps [where \(S=(S \oplus W R \ominus A L)\) ] Cons.prems
  show ?thesis
by (auto simp add: Write_{sb} True)
qed
next
case Read_{sb} with Cons show ?thesis by auto
next
case Prog_{sb} with Cons show ?thesis by auto
next
case (Ghost_{sb} A L R W)
with Cons.hyps [where $S = (S \oplus W R \ominus A L)$] Cons.prems show ?thesis by auto
qed

lemma outstanding-non-volatile-Readsb-acquired-or-read-only-reads:

$\forall \mathcal{O} S$ pending-write.

[non-volatile-owned-or-read-only pending-write $\mathcal{O} S$ sb;
a $\in$ outstanding-refs is-non-volatile-Readsb sb]

$\implies a \in$ acquired-reads True sb $\mathcal{O}$ $\vee$ a $\in$ read-only-reads $\mathcal{O}$ sb

proof (induct sb)

case Nil thus ?case by simp

next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case True
with Write sb Cons.hyps [of True $(S \oplus W R \ominus A L) (\mathcal{O} \cup A - R)$] Cons.prems
show ?thesis by auto
next
case False
with Cons show ?thesis
by (auto simp add: Write sb)
qed
next
case (Read sb volatile a' t v)
show ?thesis
proof (cases volatile)
case False with Read sb Cons show ?thesis by auto
next
case True
with Read sb Cons show ?thesis by auto
qed
next
case Progsb with Cons show ?thesis by auto
next
case (Ghost sb A L R W) with Cons.hyps [of pending-write $(S \oplus W R \ominus A L) \mathcal{O} \cup A - R]$ Cons.prems
show ?thesis
by auto
qed

lemma acquired-reads-union: $\forall$ pending-writes A B.

[a $\in$ acquired-reads pending-writes sb $(A \cup B); a \notin A$] $\implies$ a $\in$ acquired-reads pending-writes sb B

proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write \textsubscript{sb} volatile a' sop v A' L' R' W')
show ?thesis
proof (cases volatile)
case True
note volatile=this
show ?thesis
proof (cases pending-writes)
case True
from Cons. prems obtain
a-in: a \in acquired-reads True sb (A \cup B \cup A' - R') and
a-notin: a \notin A
by (simp add: Write\textsubscript{sb} volatile True)
have (A \cup B \cup A' - R') \subseteq (A \cup (B \cup A' - R'))
by auto
from acquired-reads-mono [OF this ] a-in
have a \in acquired-reads True sb (A \cup (B \cup A' - R'))
by auto

from Cons. hyps [OF this a-notin]
have a \in acquired-reads True sb (B \cup A' - R').
then show ?thesis
by (simp add: Write\textsubscript{sb} volatile True)
next
case False
with Cons show ?thesis
by (auto simp add: Write\textsubscript{sb} volatile False)
qed
next
case Read\textsubscript{sb} with Cons show ?thesis
by (auto split: if-split-asm)
next
case Prog\textsubscript{sb} with Cons show ?thesis
by (auto)
next
case (Ghost\textsubscript{sb} A' L' R' W')
show ?thesis
proof –
from Cons. prems obtain
a-in: a \in acquired-reads pending-writes sb (A \cup B \cup A' - R') and
a-notin: a \notin A
by (simp add: Ghost\textsubscript{sb} )
have \((A \cup B \cup A' - R') \subseteq (A \cup (B \cup A' - R'))\)
by auto

from acquired-reads-mono [OF this] a-in
have \(a \in\) acquired-reads pending-writes sb \((A \cup (B \cup A' - R'))\)
by auto

from Cons.hyps [OF this a-notin]
have \(a \in\) acquired-reads pending-writes sb \((B \cup A' - R')\).
then show \(\text{thesis}\)
  by (simp add: Ghostsb)
qed

lemma non-volatile-writes-unshared-no-outstanding-non-volatile-Write\(_sb\): \(\forall S S'. \ [\text{non-volatile-writes-unshared } S sb; \forall a \in \text{dom } S' - \text{dom } S. a \notin \text{outstanding-refs is-non-volatile-Write}^s_{\text{sb}} sb ] \Rightarrow \text{non-volatile-writes-unshared } S' sb\)

proof (induct sb)
case Nil thus \(\text{case}\) by simp
next
case (Cons x sb)
show \(\text{?case}\)
proof (cases x)
case (Write\(_sb\) volatile a sop v A L R W)
show \(\text{thesis}\)
proof (cases volatile)
case True
from Cons.prems obtain
unshared-sb: \(\text{non-volatile-writes-unshared } (S \oplus W R \ominus A L) sb\) and
no-refs-sb: \(\forall a \in \text{dom } S' - \text{dom } S. a \notin \text{outstanding-refs is-non-volatile-Write}^s_{\text{sb}} sb\)
by (simp add: Write\(_sb\) True)
from no-refs-sb have \(\forall a \in \text{dom } (S' \oplus W R \ominus A L) - \text{dom } (S \oplus W R \ominus A L). \ a \notin \text{outstanding-refs is-non-volatile-Write}^s_{\text{sb}} sb\)
by auto
from Cons.hyps [OF unshared-sb this] show \(\text{thesis}\)
by (simp add: Write\(_sb\) True)
next
case False
with Cons show \(\text{thesis}\)
by (auto simp add: Write\(_sb\) False)
qed
next
case Read\(_sb\) with Cons show \(\text{thesis}\)
by (auto)
next
case Prog\(_sb\) with Cons show \(\text{thesis}\)
by (auto)
next
case (Ghost_sb A L R W)
from Cons.prems obtain
  unshared-sb: non-volatile-writes-unshared \((S \oplus W R \ominus_A L)\) sb and
  no-refs-sb: \(\forall a \in \text{dom } S' - \text{dom } S\). \(a \notin \text{outstanding-refs is-non-volatile-Write}_{sb}\) sb
by (simp add: Ghost_sb)
from no-refs-sb have \(\forall a \in \text{dom } (S' \oplus W R \ominus_A L) - \text{dom } (S \oplus W R \ominus_A L)\).
  \(a \notin \text{outstanding-refs is-non-volatile-Write}_{sb}\) sb
by auto
from Cons.hyps [OF unshared-sb this]
show ?thesis
by (simp add: Ghost_sb)
qed

theorem sharing-consis-share-all-until-volatile-write:
\(\forall S\, ts'. [\text{ownership-distinct } ts; \text{sharing-consis } S ts; \text{length } ts' = \text{length } ts;\)
  \(\forall i < \text{length } ts.\)
  \(\text{let } (-, -, sb, -, O, -) = ts!i;\)
  \((-, -, sb', -, O', -) = ts'!i\)
  \(\text{in } O' = \text{acquired True (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb) } O \land\)
  \(sb' = \text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb}\)

\(\text{sharing-consis (share-all-until-volatile-write ts } S) ts' \land\)
  \(\text{dom (share-all-until-volatile-write ts } S) - \text{dom } S \subseteq \)
  \(\bigcup ((\lambda(-, -, -, sb, -, O, -). \text{set ts}) \land\)
  \(\text{dom } S - \text{dom (share-all-until-volatile-write ts } S) \subseteq \)
  \(\bigcup ((\lambda(-, -, sb, -, O, -). \text{all-acquired sb } \cup O) \iota \text{ set ts})\)

proof (induct ts)
case Nil thus ?case by auto
next
case (Cons t ts)
have leq: length ts' = length (t#ts) by fact
have sim: \(\forall i < \text{length } (t#ts).\)
  \(\text{let } (-, -, sb, -, O, -) = (t#ts)!i;\)
  \((-, -, sb', -, O', -) = ts'!i\)
  \(\text{in } O' = \text{acquired True (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb) } O \land\)
  \(sb' = \text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb}\)
by fact
obtain p is \(O R D \emptyset sb\)
  where t: t = (p, is, \emptyset, sb, D, O, R)
by (cases t)

from leq obtain t' ts'' where ts': ts' = t'#ts'' and leq': length ts'' = length ts
by (cases ts') force+

obtain p' is' O' R' D' \emptyset sb'
  where t': t' = (p', is', \emptyset, sb', D', O', R')
by (cases t')
from sim [rule-format, of 0] t t' ts'

obtain \( O' \): \( O' = \text{acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \) \( O \) and

\( sb' : sb' = \text{dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb} \)

by auto

from sim ts'

have sim': \( \forall i < \text{length ts}. \)

\( \text{(let (\(_{\sim,\sim,\sim,sb,\sim,\sim,O,\sim,R}) = \text{ts!i};} \)

\( (\_\sim,\_\sim,sb',\_\sim,O',\_\sim,R) = \text{ts''!i} \)

in \( O' = \text{acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \) \( O \land \)

\( sb' = \text{dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb} \)

by auto

have consis: sharing-consis \( S (t#ts) \) by fact

then interpret sharing-consis \( S (t#ts) \).

from sharing-consis [of 0] t

have consis-sb: sharing-consistent \( S O sb \)

by fastforce

from sharing-consistent-takeWhile [OF this]

have consis': sharing-consistent \( S O (\text{takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \)

by simp

let ?S' = (share (\text{takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \( S \))

from freshly-shared-owned [OF consis']

have fresh-owned: dom ?S' \( \subseteq O \).

from unshared-all-unshared [OF consis'] unshared-acquired-or-owned [OF consis']

have unshared-acq-owned: dom \( S - \text{dom ?S'} \)

\( \subseteq \text{all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \cup O \)

by simp

have dist: ownership-distinct (t#ts) by fact

from ownership-distinct-tl [OF this]

have dist': ownership-distinct ts .

from sharing-consis-tl [OF consis]

interpret consis': sharing-consis \( S ts \).

from dist interpret ownership-distinct (t#ts).

have sep:

\( \forall i < \text{length ts}. \text{let (\(_{\sim,\sim,\sim,sb,\sim,\sim,\sim}) = \text{ts!i in} \)

\( \text{all-acquired sb' \cap dom S - \text{dom ?S'} = \{\} \land \)

\( \text{all-unshared sb' \cap dom ?S' - \text{dom S} = \{\} \)}

proof –
\{ 
fix \text{i} \text{p}, \text{is}, \text{O}, \text{R}, \text{D}, \text{sb} \text{i} 
assume \text{i-bound: } i < \text{length ts} 
assume ts-i: ts ! i = (\text{p}, \text{is}, \text{sb}, \text{D}, \text{O}, \text{R}) 
have all-acquired sb \text{i} \cap \text{dom } S - \text{dom } ?S' = \{\} \land 
\text{all-unshared sb} \text{i} \cap \text{dom } ?S' - \text{dom } S = \{\} 
proof 
from ownership-distinct [of 0 Suc i] ts-i \text{t i-bound} 
have dist: (\text{O} \cup \text{all-acquired sb}) \cap (\text{O} \cup \text{all-acquired sb}) = \{\} 
by force 

from dist unshared-acq-owned all-acquired-takeWhile [of (Not \circ \text{is-volatile-Write_{sb}}) sb] 
have all-acquired sb \text{i} \cap \text{dom } S - \text{dom } ?S' = \{\} 
by blast 

moreover 
from sharing-consis [of Suc i] ts-i \text{i-bound} 
have sharing-consistent S \text{O}, \text{sb} 
by force 
from unshared-acquired-or-owned [OF this] 
have all-unshared sb \text{i} \subseteq \text{all-acquired sb} \cup \text{O}, 
with dist fresh-owned 
have all-unshared sb \text{i} \cap \text{dom } ?S' - \text{dom } S = \{\} 
by blast 

ultimately show \?thesis by simp 
qed 
\} 
thus \?thesis 
by (fastforce simp add: Let-def) 
qed 

from consis'.sharing-consis-preservation [OF sep] 
have consis-ts: sharing-consis \?S' ts. 

from Cons.hyps [OF dist' this leq' sim'] 
obtain consis-ts'': 
sharing-consis (share-all-until-volatile-write ts \?S') ts'' and 
fresh: dom (share-all-until-volatile-write ts \?S') - dom \?S' \subseteq 
\bigcup ((\lambda(-,-,-,-,\text{O},\text{R}). \text{O}) \text{ set ts}) and 
unshared: dom \?S' - dom (share-all-until-volatile-write ts \?S') \subseteq 
\bigcup ((\lambda(-,-,\text{sb},-\text{O},\text{R}). \text{all-acquired sb} \cup \text{O}) \text{ set ts}) 
by auto
from sharing-consistent-append \([\text{of - - (takeWhile (Not \circ is-volatile-Write_{ab}) \text{sb})} \), \(\text{dropWhile (Not \circ is-volatile-Write_{ab}) \text{sb}}\)] \(\text{consis-sb}\)

have \(\text{consis-t'}; \text{sharing-consistent} ?S' O' \text{sb'}\)
  by (simp add: \(O' \text{ sb'}\))

have \(\text{fresh-dist: all-acquired} \text{sb'} \cap \text{dom} \{S' \} - \text{dom} (\text{share-all-until-volatile-write ts} ?S') = \{\}\)
proof
  have \(\text{all-acquired} \text{sb'} \cap \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{all-acquired} \text{sb} \cup O') \text{ set ts}) = \{\}\)
  proof
    { 
      fix \(x\)
      assume \(x\)-sb': \(x \in \text{all-acquired} \text{sb'}\)
      assume \(x\)-ts: \(x \in \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{all-acquired} \text{sb} \cup O') \text{ set ts})\)
      have False
      proof
        from ownership-distinct \([\text{of 0 Suc i}] \text{ts-i} \text{i-bound}\)
        have \(\text{dist:} (O \cup \text{all-acquired sb}) \cap (O_{i} \cup \text{all-acquired sb}_{i}) = \{\}\)
        by force
        with \(x\)-sb' \(x\)-in all-acquired-dropWhile \([\text{of (Not \circ is-volatile-Write_{ab}) \text{sb}}]\) show False
        by (auto simp add: \(\text{sb'}\))
      qed
    }
  thus \(?\text{thesis by blast}\)
  qed

with unshared show \(?\text{thesis}\)
by blast
qed

have \(\text{unshared-dist: all-unshared} \text{sb'} \cap \text{dom} (\text{share-all-until-volatile-write ts} ?S') - \text{dom} \{S' = \{\}\}
proof
  from unshared-acquired-or-owned \([\text{OF \text{consis-t'}}]\)
  have \(\text{all-unshared} \text{sb'} \subseteq \text{all-acquired} \text{sb'} \cup O'\).
  also
  from all-acquired-dropWhile \([\text{of (Not \circ is-volatile-Write_{ab}) \text{sb}}]\)
  acquired-all-acquired \([\text{of True takeWhile (Not \circ is-volatile-Write_{ab}) \text{sb O}}]\)
  all-acquired-takeWhile \([\text{of (Not \circ is-volatile-Write_{ab}) \text{sb}}]\)
  have \(\text{all-acquired} \text{sb'} \cup O' \subseteq \text{all-acquired} \text{sb} \cup O\)
  by (auto simp add: \(\text{sb'} \ O'\))
  finally
  have \(\text{all-unshared} \text{sb'} \subseteq (\text{all-acquired} \text{sb} \cup O)\).

280
moreover

have \((\text{all-acquired sb} \cup \mathcal{O}) \cap \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \ \mathcal{O})^\prime \ \text{set ts}) = \{\}\)

proof –

\{ 

fix x
assume x-sb': x \in \text{all-acquired sb} \cup \mathcal{O}
assume x-ts: x \in \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \ \mathcal{O})^\prime \ \text{set ts})

have False
proof –

from x-ts
obtain i p\ i \ \text{is} \ i \ \text{O} \ i \ \text{D} \ i \ \text{θ} \ i \ \text{sb} \ i \ where
    i-bound: i < \text{length ts} \ \text{and}
    ts-i: ts!i = (p\ i, \text{is} \ i, \text{θ} \ i, \text{sb} \ i, \text{D} \ i, \text{O} \ i, \text{R} \ i) \ \text{and}
    x-in: x \in \mathcal{O}_i
by (force simp add: in-set-conv-nth)
from ownership-distinct [of 0 Suc i] ts-i t i-bound
have dist: \((\mathcal{O} \cup \text{all-acquired sb}) \cap (\mathcal{O}_i \cup \text{all-acquired sb}_i) = \{\}\)
by force
with x-sb' x-in show False
by (auto simp add: sb')
qed

\}

thus ?thesis by blast
qed

ultimately show ?thesis
using fresh by fastforce
qed

from sharing-consistent-preservation [OF consis-t' fresh-dist unshared-dist]
have consis-ts: sharing-consistent (share-all-until-volatile-write ts ?S') \mathcal{O}' sb'.

note sharing-consis-Cons [OF consis-ts' consis-ts, of p' is' \text{O}' \text{D}']

moreover
from fresh fresh-owned
have dom (share-all-until-volatile-write ts ?S') \ - \ dom \ S \subseteq \mathcal{O} \cup \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \ \mathcal{O})^\prime \ \text{set ts})
by auto
moreover
from unshared unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-Write_{sb}) sb]
have dom \ S \ - \ dom \ (share-all-until-volatile-write ts ?S') \subseteq \text{all-acquired sb} \cup \mathcal{O} \cup \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \ \text{all-acquired sb} \cup \mathcal{O})^\prime \ \text{set ts})
by auto
ultimately

show ?case
by (auto simp add: t ts' t')
qed
corollary sharing-consistent-share-all-until-volatile-write:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $S$ ts
assumes i-bound: $i < \text{length} \ ts$
assumes ts-i: $\text{ts}!i = (p,\text{is},\text{sb},D,O,R)$
shows sharing-consistent (share-all-until-volatile-write $ts$)
  
  
  \begin{align*}
  (\text{acquired True (takeWhile } \circ \text{is-volatile-Write}_{sb} \sb) O) \\
  (\text{dropWhile } \circ \text{is-volatile-Write}_{sb} \sb) \sb)
\end{align*}

proof --

define $\text{ts'}$ where $\text{ts'} == \text{map } \lambda(p,\text{is},\text{sb},D,O,R).

(p,\text{is},\theta,
  \text{dropWhile } \circ \text{is-volatile-Write}_{sb} \sb, D, \text{acquired True (takeWhile } \circ \text{is-volatile-Write}_{sb} \sb, O, R)) \text{ ts}$

have leq: length $\text{ts'} = \text{length} \ ts$
  by (simp add: $\text{ts'}$-def)

have flush: $\forall i < \text{length} \ ts$.
  
  \begin{align*}
  (\text{let } (-,-,sb',O',r) &= \text{ts}!i; \\
  (-,-,sb',O',r) &= \text{ts'}!i \\
  \text{in } O' &= \text{acquired True (takeWhile } \circ \text{is-volatile-Write}_{sb} \sb, O \land \\
  \text{sb'} &= \text{dropWhile } \circ \text{is-volatile-Write}_{sb} \sb)
\end{align*}

  by (auto simp add: $\text{ts'}$-def Let-def)

from sharing-consis-share-all-until-volatile-write [OF dist consis leq flush]
interpret sharing-consis (share-all-until-volatile-write $S$) $\text{ts'}$ by simp
from i-bound leq ts-i sharing-consis [of $i$]
show $\text{thesis}$
  by (force simp add: $\text{ts'}$-def)
qed

lemma restrict-map-UNIV [simp]: $S \mid' \text{UNIV} = S$
  by (auto simp add: restrict-map-def)

lemma share-all-until-volatile-write-Read-commute:
  
  \begin{align*}
  \land \ S \ i. \ [i < \text{length} \ ls; ls!i &= (p,\text{Read volatile a t}\#\text{is},\text{sb},D,O) &] \\
  \implies & \\
  \text{share-all-until-volatile-write (ls[i := (p,\text{is},\theta(t\rightarrow v), \text{sb}@ [\text{Read}_{ab} \text{volatile a t} v],D',O)]] S = share-all-until-volatile-write ls S)}
\end{align*}

proof (induct ls)
case Nil thus $\text{case}$
  by simp

282
next
  case (Cons l ls)
  note i-bound = \i. \i < length (l\#ls)
  note ith = \(l\#ls)!i = (p,\text{Read volatile a } t\#is,\emptyset, sb, D, O)
  show ?case
  proof (cases i)
    case 0
    from ith 0 have l: l = (p,\text{Read volatile a } t\#is,\emptyset, sb, D, O)
      by simp
    thus ?thesis
      by (simp add: 0 share-append-Read\sb del: fun-upd-apply )
  next
    case (Suc n)
    obtain p\ i\ is\ D\ \emptyset \ sb\ D' \ O\ where l: l = (p, is, \emptyset, sb, D, O)
      by (cases l)
    from i-bound ith have share-all-until-volatile-write
      (ls[n := (p, is, \emptyset, sb, D, O)]) =
      share-all-until-volatile-write ls S
      apply
      apply (rule Cons.hyps)
      apply (auto simp add: Suc l)
      done
    then
    show ?thesis
      by (simp add: Suc l del: fun-upd-apply)
  qed
qed

lemma share-all-until-volatile-write-Write-commute:
  shows \A S i.  \[ i < length ls; ls!i=(p,\text{Write volatile a } (D, f) A L R W\#is,\emptyset, sb, D, O) \]
    \implies
    share-all-until-volatile-write
      (ls[i := (p, is, \emptyset, sb, D, O)]) =
      share-all-until-volatile-write ls S
proof (induct ls)
  case Nil thus ?case
    by simp
next
  case (Cons l ls)
  note i-bound = \i. \i < length (l\#ls)
  note ith = \(l\#ls)!i = (p,\text{Write volatile a } (D, f) A L R W\#is,\emptyset, sb, D, O)
  show ?case
  proof (cases i)
    case 0
    from ith 0 have l: l = (p,\text{Write volatile a } (D, f) A L R W\#is,\emptyset, sb, D, O)
      by simp
thus \(?thesis \\
  by (simp add: share-append-Write sb del: fun-upd-apply )

next 
  case (Suc n) 
  obtain \(p_1\) \(\text{is}_1\) \(D_1\) \(\theta_1\) \(s_{b_1}\) where \(l: l = (p_1,\text{is}_1,\theta_1,s_{b_1},D_1,\text{O}_1)\) \\
  by (cases l) 
  from i-bound ith 
  have \(\text{share-all-until-volatile-write}\) 
  \(\text{(ls[n := (p_1,\text{is}_1,\theta_1,s_{b_1},D_1,\text{O}_1)])}\) 
  (share (takeWhile (Not o is-volatile-Write sb) \(s_{b_1}\)) \(S\)) = 
  \(\text{share-all-until-volatile-write ls (share (takeWhile (Not o is-volatile-Write sb) \(s_{b_1}\)) \(S\))}\) 
  apply -- 
  apply (rule Cons.hyps) 
  apply (auto simp add: Suc l) 
  done
then 
  show \(?thesis \\
  by (simp add: Suc l del: fun-upd-apply)
qed 

qed

lemma \(\text{share-all-until-volatile-write-RMW-commute}\): 
\(\forall S\ i.\ \ [i < \text{length} \ ls; \ ls!i = (p,\text{RMW a t} (D,f) \text{ cond ret A L R W#is,}\theta,\text{[]} D,\text{O})] \Rightarrow \) 
\(\text{share-all-until-volatile-write (ls[i := (p',\text{is}', \theta', \text{[]} D', \text{O'}])]) S} = \)

\(\text{share-all-until-volatile-write ls S}\)
proof 
  (induct \(ls\))
  case Nil 
  thus \(?case \\
  by simp
next 
  case (Cons l ls) 
  note i-bound = \(\langle i < \text{length} (l#ls)\rangle\) 
  note ith = \(\langle l#ls)!i = (p,\text{RMW a t} (D,f) \text{ cond ret A L R W#is,}\theta,\text{[]} D,\text{O})\rangle\) 
  show \(?case 
    proof 
    (cases i) 
    case 0 
    from ith 0 
    have \(l: l = (p,\text{RMW a t} (D,f) \text{ cond ret A L R W#is,}\theta,\text{[]} D,\text{O})\) 
    by simp 
    thus \(?thesis \\
    by (simp add: 0 share-append-Write sb del: fun-upd-apply )
next 
  case (Suc n) 
  obtain \(p_1\) \(\text{is}_1\) \(D_1\) \(\theta_1\) \(s_{b_1}\) where \(l: l = (p_1,\text{is}_1,\theta_1,s_{b_1},D_1,\text{O}_1)\) \\
  by (cases l) 
  from i-bound ith 
  have \(\text{share-all-until-volatile-write}\) 
  \(\text{(ls[n := (p',\text{is}', \theta', \text{[]} D', \text{O'}])})}\) 
  (share (takeWhile (Not o is-volatile-Write sb) \(s_{b_1}\)) \(S\)) =
share-all-until-volatile-write \( \text{ls} \) (share (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \text{S} \))

apply –
apply (rule Cons.hyps)
apply (auto simp add: Suc l)
done

then
show ?thesis
  by (simp add: Suc l del: fun-upd-apply)
qed

lemma share-all-until-volatile-write-Fence-commute:
shows \( \forall \text{i}. \ [ \text{i} < \text{length \( \text{ls} \)} ; \text{ls}[\text{i}]=(p,\text{Fence#is},\theta,[]),\langle D,O,R \rangle] \)
\( \implies \)
share-all-until-volatile-write (\text{ls}[\text{i} := (p,\text{is},\theta,[]),\langle D',O,R' \rangle]) \text{S} =
share-all-until-volatile-write \text{ls} \text{S}

proof (induct \text{ls})
  case Nil thus ?case
  by simp
next
case (Cons \text{l} \text{ls})
  note i-bound = \langle \text{i} < \text{length \( \text{l##ls} \)} \rangle
  note ith = \langle \text{l##ls}[\text{i}]=\langle p,\text{Fence#is},\theta,[]\rangle,\langle \text{D},\text{O},\text{R} \rangle \rangle
  show ?case
  proof (cases \text{i})
  case 0
  from ith 0 have \( \text{l} = \langle p,\text{Fence#is},\theta,[]\rangle,\langle \text{D},\text{O},\text{R} \rangle \)
  by simp
  thus ?thesis
  by (simp add: 0 share-append-Write_{sb} del: fun-upd-apply )
next
case (Suc \text{n})
  obtain \text{p} \text{i} \text{is} \text{O} \text{i} \text{D} \text{i} \theta \text{sb} \text{i} \text{l} \text{where} \text{l} = \langle p_{\text{i}},\text{is}_{\text{i}},\theta_{\text{i}},\text{sb}_{\text{i}},\text{D}_{\text{i}},\text{O}_{\text{i}},\text{R}_{\text{i}} \rangle
  by (cases \text{l})
  from i-bound ith
  have share-all-until-volatile-write
  (\text{ls}[\text{n} := \langle p,\text{is},\theta,[]\rangle,\langle D',O,R' \rangle])
  (share (takeWhile (Not o is-volatile-Write_{sb}) sb) \text{S}) =
share-all-until-volatile-write \text{ls} (share (takeWhile (Not o is-volatile-Write_{sb}) sb) \text{S})
  apply –
  apply (rule Cons.hyps)
  apply (auto simp add: Suc l)
done

then
show ?thesis
  by (simp add: Suc l del: fun-upd-apply)
qed
lemma unshared-share-in: \(\forall S. \ a \in \text{dom} \ S \implies a \notin \text{all-unshared} \ sb \implies a \in \text{dom} (\text{share} \ sb \ S)\)

proof (induct sb)
  case Nil thus \(\text{?case}\) by simp
next
  case (Cons x sb)
  show \(\text{?case}\)
  proof
    (cases x)
    case (Write_{sb} volatile a' sop v A L R W)
    show \(\text{?thesis}\)
    proof
      (cases volatile)
      case True
      show \(\text{?thesis}\)
      proof
        from Cons.prems obtain a-S: \(a \in \text{dom} \ S\) and a-L: \(a \notin L\) and a-sb: \(a \notin \text{all-unshared} \ sb\)
        by (clarsimp simp add: Write_{sb} True)
        from a-S a-L have \(a \in \text{dom} (S \oplus W \ominus A L)\)
        by auto
        from Cons.hyps \[\text{OF} \ this \ a-sb\]
        show \(\text{?thesis}\)
        by (clarsimp simp add: Write_{sb} True)
      qed
    next
    case False
    with Cons show \(\text{?thesis}\)
    by (auto simp add: Write_{sb} False)
    qed
  next
  case Read_{sb}
  with Cons show \(\text{?thesis}\)
  by (auto simp add: Read_{sb})
next
  case Prog_{sb}
  with Cons show \(\text{?thesis}\)
  by (auto simp add: Read_{sb})
next
  case Ghost_{sb}
  with Cons show \(\text{?thesis}\)
  by (auto simp add: Ghost_{sb})
lemma dom-eq-dom-share-eq:\ \setminus S S'. \ \text{dom} \ S = \text{dom} \ S' \implies \text{dom} \ (\text{share} \ sb \ S) = \text{dom} \ (\text{share} \ sb \ S')

proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a' sop v A' L R W)
    show ?thesis
    proof (cases volatile)
      case True
      from Cons.prems have dom (S ⊕ W R ⊖ A' L) = dom (S' ⊕ W R ⊖ A' L)
      by auto
      from Cons.hyps [OF this]
      show ?thesis by (clarsimp simp add: Write sb True)
    next
      case False with Cons.hyps [of S S'] Cons.prems Write sb show ?thesis by auto
    qed
  next
  case Read sb with Cons.hyps [of S S'] Cons.prems show ?thesis by auto
next
  case Prog sb with Cons.hyps [of S S'] Cons.prems show ?thesis by auto
next
  case (Ghost sb A' L R W)
  from Cons.prems have dom (S ⊕ W R ⊖ A' L) = dom (S' ⊕ W R ⊖ A' L)
  by auto
  from Cons.hyps [OF this]
  show ?thesis
  by (clarsimp simp add: Ghost sb)
qed
qed

lemma share-union:
\setminus A B. [a \in \text{dom} \ (\text{share} \ sb \ (A \oplus z \ B)); a \notin \text{dom} \ A] \implies a \in \text{dom} \ (\text{share} \ sb \ (\text{Map}.\text{empty} \oplus z \ B))

proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a' sop v A' L R W)
show ?thesis
proof (cases volatile)
  case True
  from Cons.prems
  obtain a-in: a ∈ dom (share sb ((A ⊕ z B) ⊕_W R ⊥_{A'}, L)) and a-A: a /∈ dom A
  by (clarsimp simp add: Write_{sb} True)
  have dom ((A ⊕ z B) ⊕_W R ⊥_{A'}, L) ⊆ dom (A ⊕ z (B ∪ R − L))
  by auto
  from share-mono [OF this] a-in
  have a ∈ dom (share sb (A ⊕ z (B ∪ R − L)))
  by blast
  moreover
  have dom (Map.empty ⊕ z (B ∪ R − L)) = dom ((Map.empty ⊕ z B) ⊕_W R ⊥_{A'}, L)
  by auto
  note dom-eq-dom-share-eq [OF this, of sb]
  ultimately
  show ?thesis
  by (clarsimp simp add: Write_{sb} True)
next
  case False
  with Cons show ?thesis
  by (auto simp add: Write_{sb} False)
qed

next
  case Read_{sb}
  with Cons show ?thesis
  by (auto simp add: Read_{sb})
next
  case Prog_{sb}
  with Cons show ?thesis
  by (auto simp add: Read_{sb})
next
  case (Ghost_{sb} A' L R W)
  from Cons.prems
  obtain a-in: a ∈ dom (share sb ((A ⊕ z B) ⊕_W R ⊥_{A'}, L)) and a-A: a /∈ dom A
  by (clarsimp simp add: Ghost_{sb})
  have dom ((A ⊕ z B) ⊕_W R ⊥_{A'}, L) ⊆ dom (A ⊕ z (B ∪ R − L))
  by auto
  from share-mono [OF this] a-in
  have a ∈ dom (share sb (A ⊕ z (B ∪ R − L)))
  by blast
  from Cons.hyps [OF this] a-A
  have a ∈ dom (share sb (Map.empty ⊕ z (B ∪ R − L)))
  by blast
  moreover
  have dom (Map.empty ⊕ z B ∪ R − L) = dom ((Map.empty ⊕ z B) ⊕_W R ⊥_{A'}, L)
  by auto

288
note dom-eq-dom-share-eq [OF this, of sb]
ultimately
show ?thesis
  by (clarsimp simp add: Ghost sb)
qed
qed

lemma share-unshared-in:
\( \forall S. a \in \text{dom} (\text{share sb } S) \implies a \in \text{dom} (\text{share sb Map.empty}) \lor (a \in \text{dom } S \land a \notin \text{all-unshared sb}) \)
proof (induct sb)
  case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
  case (Write sb volatile a' sop v A L R W)
  show ?thesis
  proof (cases volatile)
    case True
    note volatile=this
    from Cons.prems
    have a-in: a \in \text{dom} (\text{share sb } (S \oplus W R \ominus A L))
    by (clarsimp simp add: Write sb True)
    show ?thesis
    proof (cases a \in \text{dom } S)
      case True
      from share-mono-in [OF this] a-in
      have a \in \text{dom} (\text{share sb } (S \oplus W R \ominus A L))
      by blast
      from share-union [OF this False]
      show ?thesis
      proof (clarsimp simp add: Write sb volatile True)
    next
    assume a \in dom (share sb Map.empty)
    from share-mono-in [OF this]
    have a \in dom (share sb (Map.empty \oplus W R \ominus A L)) by auto
    then show ?thesis
    by (clarsimp simp add: Write sb volatile True)
  next
  assume a \in dom (S \oplus W R \ominus A L) \land a \notin \text{all-unshared sb}
  then obtain a \notin L a \notin \text{all-unshared sb}
  by auto
  then show ?thesis by (clarsimp simp add: Write sb volatile True)
qed
next
case False
have dom (S \oplus W R \ominus A L) \subseteq dom (S \oplus W (R - L))
by auto
from share-mono [OF this] a-in
have a \in dom (share sb (S \oplus W (R - L))) by blast
from share-union [OF this False]
have $a \in \text{dom} \left( \text{share}_{sb} (\text{Map.empty} \oplus_{W} (R - L)) \right)$.

moreover
have $\text{dom} (\text{Map.empty} \oplus_{W} (R - L)) = \text{dom} (\text{Map.empty} \oplus_{W} R \ominus_{A} L)$
by auto

note dom-eq-dom-share-eq [OF this, of sb]

ultimately

show $?\text{thesis}$
by (clarsimp simp add: Write_{sb} True)
qed

next

case False

with Cons show $?\text{thesis}$
by (auto simp add: Write_{sb} False)
qed

next

case Read_{sb}

with Cons show $?\text{thesis}$
by (auto simp add: Read_{sb})

next

case Prog_{sb}

with Cons show $?\text{thesis}$
by (auto simp add: Read_{sb})

next

case (Ghost_{sb} A L R W)

from Cons.prems
have $a$-in: $a \in \text{dom} \left( \text{share}_{sb} (S \oplus_{W} R \ominus_{A} L) \right)$
by (clarsimp simp add: Ghost_{sb})

show $?\text{thesis}$

proof (cases $a \in \text{dom} S$)

case True

from Cons.hyps [OF $a$-in]

have $a \in \text{dom} \left( \text{share}_{sb} \text{Map.empty} \right) \lor a \in \text{dom} (S \oplus_{W} R \ominus_{A} L) \land a \notin \text{all-unshared} \text{sb}$.

then show $?\text{thesis}$

proof

assume $a \in \text{dom} \left( \text{share}_{sb} \text{Map.empty} \right)$

from share-mono-in [OF this]

have $a \in \text{dom} \left( \text{share}_{sb} (\text{Map.empty} \oplus_{W} R \ominus_{A} L) \right)$ by auto

then show $?\text{thesis}$
by (clarsimp simp add: Ghost_{sb} True)

next

assume $a \in \text{dom} (S \oplus_{W} R \ominus_{A} L) \land a \notin \text{all-unshared} \text{sb}$

then obtain $a \notin L \land a \notin \text{all-unshared} \text{sb}$

by auto

then show $?\text{thesis}$ by (clarsimp simp add: Ghost_{sb} True)

qed

next

case False

have $\text{dom} (S \oplus_{W} R \ominus_{A} L) \subseteq \text{dom} (S \oplus_{W} (R - L))$
by auto
from share-mono [OF this] a-in
have a ∈ dom (share sb (S ⊕_W (R - L))) by blast
from share-union [OF this False]
have a ∈ dom (share sb (Map.empty ⊕_W (R - L))).
moreover
have dom (Map.empty ⊕_W (R - L)) = dom (Map.empty ⊕_W R ⊕_A L)
  by auto
note dom-eq-dom-share-eq [OF this, of sb]
ultimately
show ?thesis
  by (clarsimp simp add: Ghost_{sb} False)
qed
qed
qed

lemma dom-augment-rels-shared-eq: dom (augment-rels S R R) = dom (augment-rels S' R R)
  by (auto simp add: augment-rels-def domIff split: option.splits if-split-asm)

lemma dom-eq-SomeD1: dom m = dom n ⇒ m x = Some y ⇒ n x ≠ None
  by (auto simp add: dom-def)

lemma dom-eq-SomeD2: dom m = dom n ⇒ n x = Some y ⇒ m x ≠ None
  by (auto simp add: dom-def)

lemma dom-augment-rels-rels-eq: dom R' = dom R ⇒ dom (augment-rels S R R') = dom (augment-rels S R R)
  by (auto simp add: augment-rels-def domIff split: option.splits if-split-asm dest: dom-eq-SomeD1 dom-eq-SomeD2)

lemma dom-release-rels-eq: \( S R R' \) dom \( R' = dom R \) dom (release sb S R') = dom (release sb S R)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
hence dr: dom R' = dom R
  by simp
show ?case
proof (cases x)
case Write_{sb} with Cons.hyps [OF dr] show ?thesis by (clarsimp)
next
case Read_{sb} with Cons.hyps [OF dr] show ?thesis by (clarsimp)
next
case Prog_{sb} with Cons.hyps [OF dr] show ?thesis by (clarsimp)
next
case (Ghost_{sb} A L R W)
from Cons.hyps [OF dom-augment-rels-rels-eq [OF dr]]
show ?thesis
  by (simp add: Ghost sb)
qed
qed

lemma dom-release-shared-eq: \( \forall S S'. \) dom (release sb \( S S' \)) = dom (release sb \( S R \))
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case Write sb with Cons.hyps show ?thesis by (clarsimp)
next
case Read sb with Cons.hyps show ?thesis by (clarsimp)
next
case Prog sb with Cons.hyps show ?thesis by (clarsimp)
next
case (Ghost sb A L R W)
have dr: dom (augment-rels \( S S' R \)) = dom (augment-rels \( S R R \))
  by (rule dom-augment-rels-shared-eq)
have dom (release sb (\( S' \cup R - L \)) (augment-rels \( S' R R \))) =
  dom (release sb (\( S \cup R - L \)) (augment-rels \( S' R R \)))
  by (rule Cons.hyps)
also have ... = dom (release sb (\( S \cup R - L \)) (augment-rels \( S R R \)))
  by (rule dom-release-rels-eq [OF dr])
finally show ?thesis
  by (clarsimp simp add: Ghost sb)
qed
qed

lemma share-other-untouched:
\( \forall O S. \) sharing-consistent \( S O sb \implies a \notin O \cup all-acquired sb \implies \) share sb \( S a = S a \)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case True

from Cons.prems obtain
A-shared-owns: A \( \subseteq \) dom \( S \cup O \) and L-A: L \( \subseteq \) A and A-R: A \( \cap R = \{\} \) and R-owns: R \( \subseteq O \) and

292
consis': sharing-consistent \((S \oplus W R \ominus_A L) (O \cup A - R) sb\) and
a-owns: \(a \notin O\) and a-A: \(a \notin A\) and a-sb: \(a \notin\) all-acquired sb
by ( simp add: Write\_sb True )

\[\text{from a-owns a-A a-sb} \]
\[\text{have a \notin O \cup A - R} \cup\text{ all-acquired sb} \]
\[\text{by auto} \]
\[\text{from Cons.hyps [OF consis' this]} \]
\[\text{have share sb \((S \oplus W R \ominus_A L) a = (S \oplus W R \ominus_A L) a\).} \]
\[\text{moreover have \((S \oplus W R \ominus_A L) a = S a\)} \]
\[\text{using L-A A-R R-owns a-owns a-A} \]
\[\text{by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)} \]
\[\text{ultimately show } \text{?thesis} \]
\[\text{by (simp add: Write\_sb True)} \]

next
\[\text{case False with Cons show } \text{?thesis} \]
\[\text{by (auto simp add: Write\_sb False)} \]

qed

next
\[\text{case Read\_sb with Cons} \]
\[\text{show } \text{?thesis} \]
\[\text{by (auto)} \]

next
\[\text{case Prog\_sb with Cons} \]
\[\text{show } \text{?thesis} \]
\[\text{by (auto)} \]

next
\[\text{case (Ghost\_sb A L R W)} \]
\[\text{from Cons.prems obtain} \]
A-shared-owns: \(A \subseteq \text{dom S} \cup O\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and R-owns:
R \subseteq O and
consis': sharing-consistent \((S \oplus W R \ominus_A L) (O \cup A - R) sb\) and
a-owns: \(a \notin O\) and a-A: \(a \notin A\) and a-sb: \(a \notin\) all-acquired sb
by ( simp add: Ghost\_sb )

\[\text{from a-owns a-A a-sb} \]
\[\text{have a \notin O \cup A - R} \cup\text{ all-acquired sb} \]
\[\text{by auto} \]
\[\text{from Cons.hyps [OF consis' this]} \]
\[\text{have share sb \((S \oplus W R \ominus_A L) a = (S \oplus W R \ominus_A L) a\).} \]
\[\text{moreover have \((S \oplus W R \ominus_A L) a = S a\)} \]
\[\text{using L-A A-R R-owns a-owns a-A} \]
\[\text{by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)} \]
\[\text{ultimately show } \text{?thesis} \]
\[\text{by (simp add: Ghost\_sb)} \]

qed

qed

lemma shared-owned: \(\bigwedge O S. \) sharing-consistent \(S O sb \implies a \notin \text{dom } S \implies a \in \text{dom} (\text{share sb } S) \implies \)
\[ a \in \mathcal{O} \cup \text{all-acquired sb} \]

**proof** (induct \(sb\))

* case Nil *thus* ?case by simp*

**next**

* case (Cons \(x\) \(sb\))

* show ?case*

**proof** (cases \(x\))

* case (Write\(_{sb}\) volatile \(a'\) sop v A L R W)

* show ?thesis*

**proof** (cases volatile)

* case True*

**from** Cons,prems **obtain**

\(A\)-shared-owns: \(A \subseteq \text{dom } \mathcal{S} \cup \mathcal{O} \) \(\text{and } L\text{-A: } L \subseteq A \) \(\text{and } A\text{-R: } A \cap R = \{\} \) \(\text{and } R\text{-owns:} \)

\(R \subseteq \mathcal{O} \) \(\text{and} \)

\(\text{consis'}: \) sharing-consistent \((\mathcal{S} \oplus \mathcal{W} R \ominus_{A} L) \) \((\mathcal{O} \cup A \setminus R) \) \(\text{sb and} \)

\(\text{a-notin: } a \notin \text{dom } \mathcal{S} \) \(\text{and} \) \(a\text{-in: } a \in \text{dom } (\text{share sb } (\mathcal{S} \oplus \mathcal{W} R \ominus_{A} L)) \)

**by** (simp add: Write\(_{sb}\) True )

* show ?thesis*

**proof** (cases \(a \in \mathcal{O}\))

* case True *thus* ?thesis by auto*

**next**

* case False*

**with** a-notin R-owns A-shared-owns L-A A-R have a \( \notin \) dom \((\mathcal{S} \oplus \mathcal{W} R \ominus_{A} L) \)

**by** (auto)

**from** Cons,hyps [OF consis' this a-in]

**show** ?thesis

**by** (auto simp add: Write\(_{sb}\) True)

**qed**

**next**

* case False **with** Cons **show** ?thesis

**by** (auto simp add: Write\(_{sb}\) False)

**qed**

**next**

* case Read\(_{sb}\) **with** Cons

**show** ?thesis

**by** (auto)

**next**

* case Prog\(_{sb}\) **with** Cons

**show** ?thesis

**by** (auto)

**next**

* case (Ghost\(_{sb}\) A L R W)

**from** Cons,prems **obtain**

\(A\)-shared-owns: \(A \subseteq \text{dom } \mathcal{S} \cup \mathcal{O} \) \(\text{and } L\text{-A: } L \subseteq A \) \(\text{and } A\text{-R: } A \cap R = \{\} \) \(\text{and } R\text{-owns:} \)

\(R \subseteq \mathcal{O} \) \(\text{and} \)

\(\text{consis'}: \) sharing-consistent \((\mathcal{S} \oplus \mathcal{W} R \ominus_{A} L) \) \((\mathcal{O} \cup A \setminus R) \) \(\text{sb and} \)

\(\text{a-notin: } a \notin \text{dom } \mathcal{S} \) \(\text{and} \) \(a\text{-in: } a \in \text{dom } (\text{share sb } (\mathcal{S} \oplus \mathcal{W} R \ominus_{A} L)) \)

**by** (simp add: Ghost\(_{sb}\) )

294
show ?thesis
proof (cases a ∈ O)
  case True thus ?thesis by auto
next
  case False
  with a-notin R-owns A-shared-owns L-A have a /∈ dom (S ⊕ W R ⊖ A L)
  by (auto)
from Cons.hyps [OF consis' this a-in]
show ?thesis
  by (auto simp add: Ghostab)
qed
qed
qed

lemma share-all-shared-in: a ∈ dom (share sb S) ⟷ a ∈ dom S ∨ a ∈ all-shared sb
using sharing-consistent-share-all-shared [of sb S]
bypauto

lemma share-all-until-volatile-write-unowned:
  assumes dist: ownership-distinct ts
  assumes consis: sharing-consis S ts
  assumes other: ∀ i p is θ sb DOR. i < length ts → ts!i = (p,is,θ,sb,D,O,R) →
                   a /∈ O ∪ all-acquired sb
  shows share-all-until-volatile-write ts S a = S a
using dist consis other
proof (induct ts arbitrary: S)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  obtain pₜ isₜ Oₜ Rₜ Dₜ θₜ sbₜ where
    t: t=(pₜ,isₜ,θₜ,sbₜ,Dₜ,Oₜ,Rₜ)
  by (cases t)
from Cons.prems t obtain
  other': ∀ i p is θ sb D OR. i < length ts → ts!i = (p,is,θ,sb,D,O,R) →
          a /∈ O ∪ all-acquired sb and
          a-notin: a /∈ Oₜ ∪ all-acquired sbₜ
apply -
apply (rule that)
apply clarsimp
subgoal for i p is θ sb D OR
  apply (drule-tac x=Suc i in spec)
  apply clarsimp
done
apply (drule-tac x=0 in spec)
apply clarsimp

295
done

have dist: ownership-distinct (t#ts) by fact
then interpret ownership-distinct t#ts.
have consis: sharing-consis S (t#ts) by fact
then interpret sharing-consis S t#ts.

from ownership-distinct-tl [OF dist]
have dist’: ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis’: sharing-consis S ts.
then
interpret consis’: sharing-consis S ts.

let ?S’ = (share (takeWhile (Not ◦ is-volatile-Write sb) sb t) S)

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent S O t sb t.
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent S O t (takeWhile (Not ◦ is-volatile-Write sb) sb t).
from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom ?S’ – dom S ⊆ O t.
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: dom S – dom ?S’ ⊆ all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb t) ∪ O t
by simp

have sep:
∀ i < length ts. let (r,r,r,rb’r,r,r) = ts!i in
all-acquired sb’ ∩ dom S – dom ?S’ = {} ∧
all-unshared sb’ ∩ dom ?S’ – dom S = {}
proof –
{
fix i p i is i R i D i O i sb i
assume i-bound: i < length ts
assume ts-i: ts ! i = (p i, is i, b i, sb b i, D b i, O b i, R b i)
have all-acquired sb i ∩ dom S – dom ?S’ = {} ∧
all-unshared sb i ∩ dom ?S’ – dom S = {} ∧
proof –
from ownership-distinct [of 0 Suc i] ts-i t i-bound
have dist: (O i ∪ all-acquired sb i) ∩ (O i ∪ all-acquired sb i) = {}
by force

from dist unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-Write sb) sb t]
have all-acquired sb t ∩ dom S – dom ?S’ = {}
by blast

moreover
from sharing-consis [of Suc i] ts-i i-bound
have sharing-consistent $S \ O_i \ sb_i$
  by force
from unshared-acquired-or-owned [OF this]
have all-unshared $sb_i \subseteq$ all-acquired $sb_i \cup O_i$.
with dist fresh-owned
have all-unshared $sb_i \cap \text{dom } ?S' - \text{dom } S = \{}$
  by blast
ultimately show ?thesis by simp
  qed
}
thus ?thesis
  by (fastforce simp add: Let-def)
qed

from consis'.sharing-consis-preservation [OF this]
have sharing-consis ?S' ts.

from Cons.hyps [OF dist this other']
have share-all-until-volatile-write ts ?S' a =
  share (takeWhile (Not o is-volatile-Write$_{sb}$) sb$_t$) $S$ a .
moreover
from share-other-untouched [OF consis-sb] a-notin
  all-acquired-append [of (takeWhile (Not o is-volatile-Write$_{sb}$) sb$_t$) (dropWhile (Not o is-volatile-Write$_{sb}$) sb$_t$)]
  have share (takeWhile (Not o is-volatile-Write$_{sb}$) sb$_t$) $S$ a = $S$ a
  by auto
ultimately
show ?case
  by (simp add: t)
qed

lemma share-shared-eq: $\forall S'. S. S' a = S a \implies \text{share sb } S' a = \text{share sb } S a$
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
have eq: $S' a = S a$ by fact
show ?case
proof (cases x)
case (Write$_{sb}$ volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case True
have $(S' \oplus_w R \ominus_A L) a = (S \oplus_w R \ominus_A L) a$
using eq by (auto simp add: augment-shared-def restrict-shared-def)
from Cons.hyps [of $(S' \oplus_w R \ominus_A L)$ $(S \oplus_w R \ominus_A L)$, OF this]
show ?thesis
  by (clarsimp simp add: Write\sb True)
next
case False
  with Cons.hyps [of \S' \S] Cons.prems show ?thesis
by (auto simp add: Write\sb False)
qed
next
case Read\sb
  with Cons.hyps [of \S' \S] Cons.prems show ?thesis
  by (auto simp add: Read\sb)
next
case Prog\sb
  with Cons.hyps [of \S' \S] Cons.prems show ?thesis
  by (auto simp add: Read\sb)
next
case (Ghost\sb A L R W)
  have \((\S' \oplus W \ominus A L) a = (\S \oplus W \ominus A L) a\)
    using eq by (auto simp add: augment-shared-def restrict-shared-def)
  from Cons.hyps [of \(\S' \oplus W \ominus A L\) \(\S \oplus W \ominus A L\)] OF this
  show ?thesis
    by (clarsimp simp add: Ghost\sb)
  qed
qed

lemma share-all-until-volatile-write-thread-local:
  assumes dist: ownership-distinct ts
  assumes consis: sharing-consis \S ts
  assumes i-bound: i < length ts
  assumes ts-i: \ts!i = (p, is, \theta, sb, D, O, R)
  assumes a-owned: a \in O \cup all-acquired sb
  shows share-all-until-volatile-write ts \S a = share (takeWhile (Not \circ is-volatile-Write\sb) sb) \S a
  using dist consis i-bound ts-i
proof (induct ts arbitrary: \S i)
case Nil thus ?case by simp
next
case (Cons t ts)

obtain \pt is\t O\t R\t D\t \sigma\t sb\t where
  t: t=(\pt, is\t, \sigma\t, sb\t, D\t, O\t, R\t)
  by (cases t)

  have dist: ownership-distinct (t#ts) by fact
  then interpret ownership-distinct t#ts.
  have consis: sharing-consis \S (t#ts) by fact
  then interpret sharing-consis \S t#ts.

  from ownership-distinct-tl [OF dist]

298
have dist': ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis': sharing-consis S ts.
then
interpret consis': sharing-consis S ts.
let ?S' = (share (takeWhile (Not ◦ is-volatile-WriteSB) sb) S)

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent S O t sb.
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent S O t (takeWhile (Not ◦ is-volatile-WriteSB) sb).
from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom ?S' ⊆ O t.
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: dom S - dom ?S' ⊆ all-acquired (takeWhile (Not ◦ is-volatile-WriteSB) sb) ∪ O t

by simp

have sep:
∀i < length ts. let (·, ·, ·, sb', ·, ·, ·) = ts!i in
  all-acquired sb' ∩ dom S - dom ?S' = {} ∧
  all-unshared sb' ∩ dom ?S' - dom S = {}

proof –
  { fix i p i is i R i D i v i sb i
    assume i-bound: i < length ts
    assume ts-i: ts ! i = (p i, is i, v i, sb i, D i, O i, R i)
    have all-acquired sb i ∩ dom S - dom ?S' = {} ∧
      all-unshared sb i ∩ dom ?S' - dom S = {}
    proof –
      from ownership-distinct [of 0 Suc i] ts-i i-bound
      have dist: (O i ∪ all-acquired sb) ∩ (O i ∪ all-acquired sb) = {}
      by force

from dist unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-WriteSB) sb]
have all-acquired sb ∩ dom S - dom ?S' = {}
  by blast

moreover

from sharing-consis [of Suc i] ts-i i-bound
have sharing-consistent S O i sb
  by force
from unshared-acquired-or-owned [OF this]
have all-unshared sb ⊆ all-acquired sb ∪ O i.
with fresh-owned
have all-unshared sb ∩ dom ?S' - dom S = {}
  by blast
ultimately show ?thesis by simp 

ged 

} 
thus ?thesis

by (fastforce simp add: Let-def)

ged 

from consis'.sharing-consis-preservation [OF this]

have consis-shared': sharing-consis ?S' ts.

have aargh: (Not ◦ is-volatile-Writearb) = (λa. ¬ is-volatile-Writearb a)

by (rule ext) auto

show ?case

proof (cases i)

case 0

with Cons.prems 

have t': t = (p, is, θ, sb, D, O, R)

by simp

{

fix j p j is j θ j sb j D j O j R j

assume j-bound: j < length ts

assume ts-j: ts ! j = (p j, is j, θ j, sb j, D j, O j, R j)

have a ∈ O j ∪ all-acquired sb j

proof –

from ownership-distinct [of 0 Suc j, simplified, OF j-bound t ts-j] t a-owned t' 0

show ?thesis

by auto

ged 

}

with share-all-until-volatile-write-unowned [OF dist' consis-shared', of a]

have share-all-until-volatile-write ts ?S' a = ?S' a

by fastforce

then show ?thesis

using t' 0

by (auto simp add: Cons t aargh)

next

case (Suc n)

with Cons.prems obtain n-bound: n < length ts and ts-n: ts!n = (p, is, θ, sb, D, O, R)

by auto

from Cons.hyps [OF dist' consis-shared' n-bound ts-n]

have share-all-until-volatile-write ts ?S' a =

share (takeWhile (Not ◦ is-volatile-Writearb) sb) ?S' a .

moreover

from ownership-distinct [of 0 Suc n] t a-owned ts-n n-bound

have a ∈ O t ∪ all-acquired sb t
by fastforce
with share-other-untouched [OF consis-sb, of a]
all-acquired-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb) (dropWhile (Not ◦ is-volatile-Write sb) sb)]
have ?S' a = S a
by auto
from share-shared-eq [of ?S' a S, OF this]
have share (takeWhile (Not ◦ is-volatile-Write sb) sb) ?S' a = share (takeWhile (Not ◦ is-volatile-Write sb) sb) S a.
ultimately show ?thesis
using t Suc
by (auto simp add: aargh)
qed

lemma share-all-until-volatile-write-thread-local':
assumes dist: ownership-distinct ts
assumes consis: sharing-consis S ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb, D, O, R)
assumes a-owned: a ∈ O ∪ all-acquired sb
shows share (dropWhile (Not ◦ is-volatile-Write sb) sb) (share-all-until-volatile-write ts S) a = share sb S a
proof
−
let ?take = takeWhile (Not ◦ is-volatile-Write sb) sb
let ?drop = dropWhile (Not ◦ is-volatile-Write sb) sb
from share-all-until-volatile-write-thread-local [OF dist consis i-bound ts-i a-owned]
have share-all-until-volatile-write ts S a = share ?take S a.
moreover
from share-shared-eq [of share-all-until-volatile-write ts S a share ?take S, OF this]
have share ?drop (share-all-until-volatile-write ts S) a = share ?drop (share ?take S) a.
thus ?thesis
using share-append [of ?take ?drop S]
by simp
qed

lemma (in ownership-distinct) in-shared-sb-share-all-until-volatile-write:
assumes consis: sharing-consis S ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb, D, O, R)
assumes a-owned: a ∈ O ∪ all-acquired sb
assumes a-share: a ∈ dom (share sb S)
shows a ∈ dom (share (dropWhile (Not ◦ is-volatile-Write sb) sb) (share-all-until-volatile-write ts S))
proof
have dist: ownership-distinct ts
using assms ownership-distinct
apply −
apply (rule ownership-distinct.intro)
apply auto
done
from share-all-until-volatile-write-thread-local' [OF dist consis i-bound ts-i a-owned]
a-share
show ?thesis
  by (auto simp add: domIff)
qed

lemma owns-unshared-share-acquired:
  \( \forall \mathcal{O}. \) [sharing-consistent \( \mathcal{S} \mathcal{O} \) \( sb; a \in \mathcal{O}; a \notin \text{all-unshared} \) \( sb \) ]
  \( \implies a \in \text{dom} (\text{share} sb \mathcal{S}) \cup \text{acquired True} sb \mathcal{O} \)
proof (induct \( sb \))
case Nil thus ?case by auto
next
case (Cons \( x \) \( sb \))
show ?case
proof (cases \( x \))
case (Write\( sb \) volatile \( a' \) sop v A L R W)
show ?thesis
proof (cases volatile)
case True
note volatile=this
from Cons.prems obtain
a-owns: \( a \in \mathcal{O} \) and A-shared-onws: \( A \subseteq \text{dom} \mathcal{S} \cup \mathcal{O} \) and
a-L: \( a \notin L \) and a-unsh: \( a \notin \text{all-unshared} sb \) and L-A: \( L \subseteq A \) and
A-R: \( A \cap R = \{\} \) and R-owns: \( R \subseteq \mathcal{O} \) and
consis': sharing-consistent (\( \mathcal{S} \oplus_w R \ominus_A L \)) \( (\mathcal{O} \cup A - R) \) sb
by (clarsimp simp add: Write\( sb \) volatile)
  have a \in dom (\text{share} sb (\( \mathcal{S} \oplus_w R \ominus_A L \)) \cup \text{acquired True} sb (\( \mathcal{O} \cup A - R \))
  proof (cases a \in R)
case True
  with a-L have a \in dom (\( \mathcal{S} \oplus_w R \ominus_A L \))
  by auto
  from unshared-share-in [OF this a-unsh]
  show ?thesis by blast
next
case False
hence a \in \( \mathcal{O} \cup A - R \)
  using a-owns
  by auto
  from Cons.hyps [OF consis' this a-unsh]
  show ?thesis .
  qed
  then
  show ?thesis
  by (clarsimp simp add: Write\( sb \) volatile)
next
case False
  with Cons
  show ?thesis

302
by (auto simp add: Write) 
qed

next
case Read
with Cons show thesis 
by (auto simp add: Read)

next
case Progsb
with Cons show thesis 
by (auto simp add: Read)

next
case (Ghostsb A L R W)
from Cons.prems obtain 
a-owns: a ∈ O and A-shared-onws: A ⊆ dom S ∪ O and 
a-L: a /∈ L and a-unsh: a /∈ all-unshared sb and L-A: L ⊆ A and 
A-R: A ∩ R = {} and R-owns: R ⊆ O and 
consis': sharing-consistent (S ⊕ R R ⊃ A L) (O ∪ A − R) sb 
by (clarsimp simp add: Ghost)

have a ∈ dom (share sb (S ⊕ R R ⊃ A L)) ∪ acquired True sb (O ∪ A − R)

proof (cases a ∈ R)
case True 
with a-L have a ∈ dom (S ⊕ R R ⊃ A L) 
by auto
from unshared-share-in [OF this a-unsh]

show thesis by blast

next
case False 
hence a ∈ O ∪ A − R

using a-owns
by auto
from Cons.hyps [OF consis' this a-unsh]

show thesis .

qed
then show thesis 
by (auto simp add: Ghost)

qed

lemma shared-share-acquired: \(\bigwedge S O. \text{ sharing-consistent } S O sb \implies a \in \text{ dom } S \implies a \in \text{ dom } (\text{share sb } S) \cup \text{ acquired True sb } O\)

proof (induct sb)
case Nil thus case by auto

next
case (Cons x sb)
show case 

proof (cases x)
case (Write volatile a' sop v A L R W)
show thesis 

proof (cases volatile)
case True
note volatile=this
from Cons.prems obtain
a-shared: a ∈ dom S and A-shared-owns: A ⊆ dom S ∪ O and
L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
consis': sharing-consistent (S ⊕ W R ⊖ A L) (O ∪ A − R) sb
by (clarsimp simp add: Write sb True)
show ?thesis
proof (cases a ∈ L)
case False with a-shared
have a ∈ dom (S ⊕ W R ⊖ A L)
by auto
from Cons.hyps [OF consis' this]
show ?thesis
by (clarsimp simp add: Write sb volatile)
next
case True
with L-A have a-A: a ∈ A
by blast
from sharing-consistent-mono-shared [OF - consis', where S'=(S ⊕ W R)]
have sharing-consistent (S ⊕ W R) (O ∪ A − R) sb
by auto
from Cons.hyps [OF this] a-shared
have hyp: a ∈ dom (share sb (S ⊕ W R)) ∪ acquired True sb (O ∪ A − R)
by auto
{
assume a ∈ dom (share sb (S ⊕ W R))
from share-unshared-in [OF this]
have a ∈ dom (share sb (S ⊕ W R ⊖ A L)) ∪ acquired True sb (O ∪ A − R)
proof
assume a ∈ dom (share sb Map.empty)
from share-mono-in [OF this]
have a ∈ dom (share sb (S ⊕ W R ⊖ A L))
by auto
thus ?thesis by blast
next
assume a ∈ dom (S ⊕ W R) ∧ a ∉ all-unshared sb
hence a-unsh: a ∉ all-unshared sb by blast
from a-A A-R have a ∈ O ∪ A − R
by auto
from owns-unshared-share-acquired [OF consis' this a-unsh]
show ?thesis .
qed
}
with hyp show ?thesis
by (auto simp add: Write sb volatile)
qed
next
case False
with Cons
show ?thesis
by (auto simp add: Write\$_{ab}")

qed

next

case Read\$_{ab}

with Cons show \?(thesis

by (auto simp add: Read\$_{ab})

next

case Prog\$_{ab}

with Cons show \?(thesis

by (auto simp add: Read\$_{ab})

next

case (Ghost\$_{ab} A L R W)

from Cons.prems obtain

a-shared: a \in dom \(S) and A-shared-owns: A \subseteq dom \(S \cup O) and

L-A: L \subseteq A and A-R: A \cap R = \{\} and R-owns: R \subseteq O and

consis': sharing-consistent \((S \oplus W R \ominus A L) (O \cup A – R) sb

by (clarsimp simp add: Ghost\$_{ab})

show \?(thesis

proof (cases a \in L)

case False with a-shared

have a \in dom \((S \oplus W R \ominus A L)

by auto

from Cons.hyps [OF consis’ this]

show \?(thesis

by (clarsimp simp add: Ghost\$_{ab})

next

case True

with L-A have a-A: a \in A

by blast

from sharing-consistent-mono-shared [OF - consis’, where \(S’=(S \oplus W R)]

have sharing-consistent \((S \oplus W R) (O \cup A – R) sb

by auto

from Cons.hyps [OF this] a-shared

have hyp: a \in dom (share sb \((S \oplus W R)) \cup acquired True sb (O \cup A – R)

by auto

{ assume a \in dom (share sb \((S \oplus W R))

from share-unshared-in [OF this]

have a \in dom (share sb \((S \oplus W R \ominus A L)) \cup acquired True sb (O \cup A – R)

proof

assume a \in dom (share sb Map.empty)

from share-mono-in [OF this]

have a \in dom (share sb \((S \oplus W R \ominus A L))

by auto

thus \?(thesis by blast

next

assume a \in dom \((S \oplus W R) \land a \notin all-unshared sb

hence a-unsh: a \notin all-unshared sb by blast

from a-A A-R have a \in O \cup A – R

by auto

305
show \thesis .
qed \}

with hyp show \thesis
by (auto simp add: Ghost sb)
qed
qed

lemma dom-release-takeWhile:

\[ S \mathcal{R}. \]
\begin{align*}
\text{dom (release (takeWhile (Not o is-volatile-Write sb) sb) S \mathcal{R})} &= \text{dom } \mathcal{R} \cup \text{all-shared (takeWhile (Not o is-volatile-Write sb) sb)}
\end{align*}

apply (induct sb)
apply (clarsimp)
subgoal for a sb S \mathcal{R}
apply (case-tac a)
apply (auto simp add: augment-rels-def domIff split: if-split-asm option.splits)
done
done

lemma share-all-until-volatile-write-share-acquired:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis S ts
assumes a-notin: a \notin dom S
assumes a-in: a \in dom (share-all-until-volatile-write ts S)
shows \exists i < \text{length ts}.
\begin{align*}
\text{let (\_\_\_\_, \_\_\_) = ts!i}
\text{in a \in all-shared (takeWhile (Not o is-volatile-Write sb) sb)}
\end{align*}

using dist consis a-notin a-in
proof (induct ts arbitrary: S i)
case Nil thus \?case by simp
next
case (Cons t ts)

have a-notin: a \notin dom S by fact
obtain p_{t} is_{t} O_{t} D_{t} v_{t} sb_{t} \text{ where}
\begin{align*}
t &= (p_{t}, is_{t}, v_{t}, sb_{t}, D_{t}, O_{t}, \mathcal{R}_{t})
\end{align*}
by (cases t)

let ?take = (takeWhile (Not o is-volatile-Write sb) sb)
from t Cons.prems
have a-in: a \in dom (share-all-until-volatile-write ts (share ?take S))
by auto

have dist: ownership-distinct (t#ts) by fact
then interpret ownership-distinct t#ts.
have consis: sharing-consis S (t#ts) by fact
then interpret sharing-consis S t#ts.
from ownership-distinct-tl [OF dist]
have dist': ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis': sharing-consis S ts.
then
interpret consis': sharing-consis S ts.
let ?S' = (share ?take S)

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent S O_t sb_t.
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent S O_t ?take.
from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom ?S' − dom S ⊆ O_t.
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: dom S − dom ?S' ⊆ all-acquired ?take ∪ O_t
by simp

have sep:
∀ i < length ts. let (.,.,.,sb'_i,.,.,.) = ts'_i in
  all-acquired sb'_i ∩ dom S − dom ?S' = {} ∧
  all-unshared sb'_i ∩ dom ?S' − dom S = {}

proof –
{ 
  fix i p_i is_i R_i D_i \emptyset_i sb_i
  assume i-bound: i < length ts
  assume ts-i: ts ! i = (p_i, is_i, \emptyset_i, sb_i, D_i, O_i, R_i)
  have all-acquired sb_i ∩ dom S − dom ?S' = {} ∧
         all-unshared sb_i ∩ dom ?S' − dom S = {} 
  proof –
  from ownership-distinct [of 0 Suc i] ts-i t i-bound
  have dist: (O_t ∪ all-acquired sb_t) ∩ (O_t ∪ all-acquired sb_i) = {}
  by force

from dist unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-Write sb) sb_i]
have all-acquired sb_i ∩ dom S − dom ?S' = {}
by blast

moreover

from sharing-consis [of Suc i] ts-i i-bound
have sharing-consistent S O_i sb_i
by force
from unshared-acquired-or-owned [OF this]
have all-unshared sb_i ⊆ all-acquired sb_i ∪ O_i,
with dist fresh-owned

307
have all-unshared sb \_1 \cap \text{dom } ?S' - \text{dom } S = \{\}

by blast

ultimately show \?thesis by simp

qed

\}

thus \?thesis

by (fastforce simp add: Let-def)

qed

from consis'.sharing-consis-preservation [OF this]

have consis-shared': sharing-consis ?S' ts.

have aargh: (Not \circ is-volatile-Write_{ab}) = (\lambda a. \neg is-volatile-Write_{ab} a)

by (rule ext) auto

show \?case

proof (cases a \in all-shared ?take)

  case True

  thus \?thesis

  apply -

  apply (rule-tac x=0 in exI)

  apply (auto simp add: t aargh)

  done

next

  case False

have a-notin': a \notin \text{dom } ?S'

proof

  assume a \in \text{dom } ?S'

  from share-all-shared-in [OF this] False a-notin

  show False

  by auto

qed

from Cons.hyps [OF dist' consis-shared' a-notin'a-in]

obtain i where i < length ts and

  rel: let (p,is,0,\_sb,D,O,R) = ts\_i

  in a \in all-shared (takeWhile (Not \circ is-volatile-Write_{ab}) sb)

  by (auto simp add: Let-def aargh)

then show \?thesis

  apply -

  apply (rule-tac x = Suc i in exI)

  apply (auto simp add: Let-def aargh)

  done

qed

qed

lemma all-shared-share-acquired: \bigwedge S O. sharing-consistent S O sb \implies

a \in all-shared sb \implies a \in \text{dom } (\text{share sb } S) \cup \text{acquired True sb } O
proof (induct sb)
case Nil thus ?case by auto
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case True
note volatile=this
from Cons.prems obtain
a-shared: a ∈ R \cup all-shared sb and A-shared-owns: A ⊆ dom S \cup O and
L-A: L ⊆ A and A-R: A \cap R = {} and R-owns: R ⊆ O and
consis': sharing-consistent (S ⊕ R ⊖ A L) (O \cup A - R) sb
by (clarsimp simp add: Write sb True)
show ?thesis
proof (cases a ∈ all-shared sb)
case True
from Cons.hyps [OF consis' True]
show ?thesis
  by (clarsimp simp add: Write sb volatile)
next
case False
with a-shared have a ∈ R
  by auto
with L-A A-R R-owns have a ∈ dom (S ⊕ W R ⊖ A L)
  by auto
from shared-share-acquired [OF consis' this]
show ?thesis
  by (clarsimp simp add: Write sb volatile)
qed
next
case False
with Cons show ?thesis
  by (auto simp add: Write sb)
qed
next
case Read sb
with Cons show ?thesis
  by (auto simp add: Read sb)
next
case Prog sb
with Cons show ?thesis
  by (auto simp add: Read sb)
next
case (Ghost sb A L R W)
from Cons.prems obtain
a-shared: a ∈ R \cup all-shared sb and A-shared-owns: A ⊆ dom S \cup O and
L-A: \( L \subseteq A \) and A-R: \( A \cap R = \{\} \) and R-owns: \( R \subseteq O \) and
consis': sharing-consistent \( (S \oplus W R \ominus A L) (O \cup A - R) \) sb
by (clarsimp simp add: Ghost\(_{sb}\))
show \(?thesis
proof (cases a \( \in \) all-shared sb)
  case True
  from Cons.hyps [OF consis' True]
  show \(?thesis
    by (clarsimp simp add: Ghost\(_{sb}\))
next
  case False
  with a-shared have a \( \in \) R
  by auto
  with L-A A-R R-owns have a \( \in \) dom \( (S \oplus W R \ominus A L) \)
  by auto
  from shared-share-acquired [OF consis' this]
  show \(?thesis
    by (clarsimp simp add: Ghost\(_{sb}\))
qed
qed

lemma (in ownership-distinct) share-all-until-volatile-write-share-acquired:
assumes consis: sharing-consis \( S \) ts
assumes i-bound: \( i < \) length ts
assumes ts-i: ts\(!i\) = (p,\(i\),\(0\),sb,\(D\),O,R)
assumes a-in: a \( \in \) dom (share-all-until-volatile-write ts \( S \))
sshows a \( \in \) dom (share sb \( S \)) \( \vee \) a \( \in \) acquired True sb \( O \) \( \vee \)
(\( \exists j < \) length ts. \( j \neq i \wedge \)
  (let \((r,\mathcal{r},sb,j,r,\mathcal{r}) = ts\!j\)
    in a \( \in \) all-shared (takeWhile (Not \( \circ \) is-volatile-Write\(_{sb}\)) sb\(_j\))))
proof –
from assms ownership-distinct have dist: ownership-distinct ts
apply –
apply (rule ownership-distinct.intro)
apply simp
done
from consis
interpret sharing-consis \( S \) ts .
from sharing-consis [OF i-bound ts-i]
have consis-sb: sharing-consistent \( S \) \( O \) sb.

let ?take-sb = takeWhile (Not \( \circ \) is-volatile-Write\(_{sb}\)) sb
let ?drop-sb = dropWhile (Not \( \circ \) is-volatile-Write\(_{sb}\)) sb

show \(?thesis
proof (cases a \( \in \) dom \( S \))
  case True
  from shared-share-acquired [OF consis-sb True]
  have a \( \in \) dom (share sb \( S \)) \( \cup \) acquired True sb \( O \).
thus ?thesis by auto
next
case False
from share-all-until-volatile-write-share-acquired [OF dist consis False a-in]
obtain j where j-bound: j < length ts and
  rel: let (\_,\_,sb_j,\_,\_,\_) = ts\_j
  in a ∈ all-shared (takeWhile (Not o is-volatile-Write\_sb) sb_j)
  by auto
show ?thesis
proof (cases j=i)
case False
with j-bound rel
show ?thesis
  by blast
next
case True
with rel ts-i have a ∈ all-shared ?take-sb
  by (auto simp add: Let-def)
hence a ∈ all-shared sb
using all-shared-append [of ?take-sb ?drop-sb]
  by auto
from all-shared-share-acquired [OF consis-sb this]
have a ∈ dom (share sb S) ∪ acquired True sb O.
thus ?thesis
  by auto
qed
qed
qed

lemma acquired-all-shared-in:
\( \wedge A. \ a ∈ acquired \ True \ sb \ A \implies a ∈ acquired \ True \ sb \ \{\} \lor (a ∈ A \land a \notin \ all-shared \ sb) \)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write\_sb volatile a’ sop v A’ L R)
show ?thesis
proof (cases volatile)
case True
note volatile=this
from Cons.prems
have a-in: a ∈ acquired True sb (A ∪ A’ − R)
by (clarsimp simp add: Write\_sb True)
show ?thesis
proof (cases a ∈ A)
case True
from Cons.hyps [OF a-in]
have a ∈ acquired True sb {} ∨ a ∈ A ∪ A′ − R ∧ a /∈ all-shared sb.
them show ?thesis
proof
assume a ∈ acquired True sb {}
from acquired-mono-in [OF this]
have a ∈ acquired True sb (A′ − R) by auto
then show ?thesis
  by (clarsimp simp add: Write sb volatile True)
next
assume a ∈ A ∪ A′ − R ∧ a /∈ all-shared sb
then obtain a /∈ R a /∈ all-shared sb
  by blast
then show ?thesis by (clarsimp simp add: Write sb volatile True)
qed
next
case False
have (A ∪ A′ − R) ⊆ A ∪ (A′ − R)
  by blast
from acquired-mono [OF this] a-in
have a ∈ acquired True sb (A ∪ (A′ − R)) by blast
from acquired-union-notin-first [OF this False]
have a ∈ acquired True sb (A′ − R).
then show ?thesis
  by (clarsimp simp add: Write sb True)
qed
next
case False
with Cons show ?thesis
by (auto simp add: Write sb False)
qed
next
case Read sb
with Cons show ?thesis
  by (auto simp add: Read sb)
next
case Prog sb
with Cons show ?thesis
  by (auto simp add: Read sb)
next
case (Ghost sb A′ L R W)
from Cons.prems
have a-in: a ∈ acquired True sb (A ∪ A′ − R)
  by (clarsimp simp add: Ghost sb)
show ?thesis
proof (cases a ∈ A)
case True

312
from Cons.hyps [OF a-in]
have a ∈ acquired True sb { } ∨ a ∈ A ∪ A’ − R ∧ a /∈ all-shared sb.
thен show ?thesis
proof
assume a ∈ acquired True sb { }
from acquired-mono-in [OF this]
have a ∈ acquired True sb (A’ − R) by auto
then show ?thesis
by (clarsimp simp add: Ghostsb True)
next
assume a ∈ A ∪ A’ − R ∧ a /∈ all-shared sb
then obtain a /∈ R a /∈ all-shared sb
by blast
then show ?thesis by (clarsimp simp add: Ghostsb True)
qed

next
case False
have (A ∪ A’ − R) ⊆ A ∪ (A’ − R)
by blast
from acquired-mono [OF this] a-in
have a ∈ acquired True sb (A ∪ (A’ − R)) by blast
from acquired-union-notin-first [OF this False]
have a ∈ acquired True sb (A’ − R).
then show ?thesis
by (clarsimp simp add: Ghostsb)
qed
qed

lemma all-shared-acquired-in: ∃ A. a ∈ A =⇒ a /∈ all-shared sb =⇒ a ∈ acquired True sb A
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a’ sop v A’ L R W)
show ?thesis
proof (cases volatile)
case True
show ?thesis
proof –
from Cons.prems obtain a-A: a ∈ A and a-R: a /∈ R and a-sb: a /∈ all-shared sb
by (clarsimp simp add: Writelsb True)
from a-A a-R have a ∈ A ∪ A’ − R
by blast
from Cons.hyps [OF this a-sb]
show ?thesis
by (clarsimp simp add: Write sb True)
qed

next

  case False
  with Cons show ?thesis
  by (auto simp add: Write sb False)
qed

next

  case Read sb
  with Cons show ?thesis
  by (auto simp add: Read sb)

next

  case Prog sb
  with Cons show ?thesis
  by (auto simp add: Read sb)

next

  case Ghost sb
  with Cons show ?thesis
  by (auto simp add: Ghost sb)

qed

lemma owned-share-acquired: \( \bigwedge \mathcal{S} \mathcal{O} \). sharing-consistent \( \mathcal{S} \mathcal{O} \) \( \Rightarrow \) \( a \in \mathcal{O} \Rightarrow a \in \text{dom} (\text{share } \mathcal{S} \cup \text{acquired } \text{True } \mathcal{S} \cup \mathcal{O}) \)
proof (induct sb)
  case Nil thus ?case by auto
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a’ sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True
      note volatile=this
      from Cons.prems obtain
      a-owned: \( a \in \mathcal{O} \) and A-shared-owns: \( A \subseteq \text{dom } \mathcal{S} \cup \mathcal{O} \) and
      L-A: \( L \subseteq A \) and A-R: \( A \cap R = \{\} \) and R-owns: \( R \subseteq \mathcal{O} \) and
      consis’: sharing-consistent \( \langle \mathcal{S} \oplus_W R \ominus_A L \rangle \) \( \langle \mathcal{O} \cup A - R \rangle \) sb
      by (clarsimp simp add: Write sb True)
      show ?thesis
      proof (cases a \in R)
      case False with a-owned
      have a \in \mathcal{O} \cup A - R
      by auto
      from Cons.hyps [OF consis’ this]
      show ?thesis
      proof (clarsimp simp add: Write sb volatile)
      next
    case True
  qed

314
from True L-A A-R have a ∈ dom (S ⊕ W R ⊖ A L)
  by auto
from shared-share-acquired [OF consis′ this]
show ?thesis
  by (clarsimp simp add: Write sb volatile True)
  qed
next
case False
with Cons
  show ?thesis
  by (auto simp add: Write sb)
  qed
next
case Read sb
with Cons
  show ?thesis
  by (auto simp add: Read sb)
  qed
next
case Prog sb
with Cons
  show ?thesis
  by (auto simp add: Read sb)
next
case (Ghost sb A L R W)
from Cons.prems obtain
  a-owned: a ∈ O and A-shared-owns: A ⊆ dom S ∪ O and
  L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
  consis′: sharing-consistent (S ⊕ W R ⊖ A L) (O ∪ A − R) sb
  by (clarsimp simp add: Ghost sb)
show ?thesis
proof (cases a ∈ R)
case False with a-owned
  have a ∈ O ∪ A − R
    by auto
from Cons.hyps [OF consis′ this]
  show ?thesis
    by (clarsimp simp add: Ghost sb)
next
case True
from True L-A A-R have a ∈ dom (S ⊕ W R ⊖ A L)
  by auto
from shared-share-acquired [OF consis′ this]
  show ?thesis
    by (clarsimp simp add: Ghost sb True)
  qed
qed
qed

lemma outstanding-refs-non-volatile-Read sb-all-acquired:
\∧ m S O pending-write.
reads-consistent pending-write $O \ m s b; \text{non-volatile-owned-or-read-only pending-write}
$S O s b$
a $\in$ outstanding-refs is-non-volatile-Read$_{sb}$ $s b$
$\implies a \in O \lor a \in \text{all-acquired } s b \lor$
a $\in$ read-only-reads $O s b$

proof (induct $s b$)
case Nil thus ?case by simp
next
case (Cons x $s b$)
show ?case
proof (cases x)
case (Write$_{sb}$ volatile $a^\prime$ sop v A L R W)
show ?thesis
proof (cases volatile)
case True
note volatile=this from Cons.prems obtain
non-vo: non-volatile-owned-or-read-only True ($S \oplus W R \ominus A L$)
($O \cup A - R$) $s b$ and
out-vol: outstanding-refs is-volatile-Read$_{sb}$ $s b$ = {} and
out: a $\in$ outstanding-refs is-non-volatile-Read$_{sb}$ $s b$
by (clarsimp simp add: Write$_{sb}$ True)
show ?thesis
proof (cases a $\in O$)
case True
show ?thesis
by (clarsimp simp add: Write$_{sb}$ True volatile)
next
case False
from outstanding-non-volatile-Read$_{sb}$-acquired-or-read-only-reads [OF non-vo out]
have a-in: a $\in$ acquired-reads True $s b$ ($O \cup A - R$) \lor
a $\in$ read-only-reads ($O \cup A - R$) $s b$
by auto
with acquired-reads-all-acquired [of True $s b$ ($O \cup A - R$)]
show ?thesis
by (auto simp add: Write$_{sb}$ False)
qed
next
case Read$_{sb}$
with Cons show ?thesis
apply (clarsimp simp del: o-apply simp add: Read$_{sb}$ acquired-takeWhile-non-volatile-Write$_{sb}$ split: if-split-asm)
apply auto
done
next
case Progsb
  with Cons show ?thesis
  by (auto simp add: Read$_{sb}$)
next
  case (Ghost$_{sb}$ A L)
  with Cons show ?thesis
  by (auto simp add: Ghost$_{sb}$)
qed
qed

lemma outstanding-refs-non-volatile-Read$_{sb}$-all-acquired-dropWhile:
assumes consis: reads-consistent pending-write $S \ O \ sb$
assumes nvo: non-volatile-owned-or-read-only pending-write $S \ O \ sb$
assumes out: a $\in$ outstanding-refs is-non-volatile-Read$_{sb}$ (dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)
shows a $\in$ $O \lor$ a $\in$ all-acquired sb $\lor$
a $\in$ read-only-reads $O \ sb$
using outstanding-refs-append $[\text{of is-non-volatile-Read}_{sb}$ takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb
  dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb]
outstanding-refs-non-volatile-Read$_{sb}$-all-acquired $[\text{OF consis nvo, of a}]$ out
by (auto)

lemma share-commute:
$[L \ R \ S \ O]. \ [\text{sharing-consistent} \ S \ O \ sb];$
all-shared sb $\cap$ L = {}; all-shared sb $\cap$ A = {}; all-acquired sb $\cap$ R = {};
all-unshared sb $\cap$ R = {}; all-shared sb $\cap$ R = {}$\Rightarrow$
(share sb $(S \oplus_{W} R \ominus_{A} L)) =$
(share sb $S)$ $\oplus_{W} R \ominus_{A} L$
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write$_{sb}$ volatile a sop v A’ L’ R’ W’)
show ?thesis
proof (cases volatile)
  case True
  note volatile=this
  from Cons.prems obtain
  L-prop: (R’ $\cup$ all-shared sb) $\cap$ L = {} and
  A-prop: (R’ $\cup$ all-shared sb) $\cap$ A = {} and
  R-acq-prop: (A’ $\cup$ all-acquired sb) $\cap$ R = {} and
  R-prop: (L’ $\cup$ all-unshared sb) $\cap$ R = {} and
  R-prop-sh: (R’ $\cup$ all-shared sb) $\cap$ R = {} and
  A’-shared-owns: A’ $\subseteq$ dom $S \cup O$ and L’-A’: L’ $\subseteq$ A’ and A’-R’: A’ $\cap$ R’ = {} and
R′-owns: R′ ⊆ O and 

consis': sharing-consistent (S ⊕ W′ R′ ⊆A′ L′) (O ∪ A′ - R′) sb 

by (clarsimp simp add: Write sb volatile)

from L-prop obtain R′-L: R′ ∩ L = {} and acq-L: all-shared sb ∩ L = {}
by blast
from A-prop obtain R′-A: R′ ∩ A = {} and acq-A: all-shared sb ∩ A = {}
by blast
from R-acq-prop obtain A′-R: A′ ∩ R = {} and acq-R: all-acquired sb ∩ R = {}
by blast
from R-prop obtain L′-R: L′ ∩ R = {} and unsh-R: all-unshared sb ∩ R = {}
by blast
from R-prop-sh obtain R′-R: R′ ∩ R = {} and sh-R: all-shared sb ∩ R = {}
by blast

from Cons.hyps [OF consis’ acq-L acq-A acq-R unsh-R sh-R ]
have share sb ((S ⊕ W′ R′ ⊆A′ L′) ⊆W R ⊆A L) = share sb (S ⊕ W′ R′ ⊆A′ L′) ⊆W R ⊆A L.

moreover

from R′-L L′-R R′-R R′-A A′-R
have ((S ⊕ W R ⊆A L) ⊆W R′ ⊆A′ L′) = ((S ⊕ W′ R′ ⊆A′ L′) ⊆W R ⊆A L)
apply –
apply (clarsimp simp add: augment-shared-def restrict-shared-def)
apply (auto split: if-split-asm option.splits)
done

ultimately

have share sb ((S ⊕ W R ⊆A L) ⊆W R′ ⊆A′ L′) = share sb (S ⊕ W′ R′ ⊆A′ L′) ⊆W R ⊆A L
by simp
then
show ?thesis
by (clarsimp simp add: Write sb volatile)
next
case False with Cons show ?thesis
by (clarsimp simp add: Write sb False)
qed
next
case Read sb with Cons show ?thesis
by (clarsimp simp add: Read sb)
next
case Prog sb with Cons show ?thesis
by (clarsimp simp add: Prog sb)
next
case (Ghost sb A′ L′ R′ W′)
from Cons.prems obtain
L-prop: \( (R' \cup \text{all-shared sb}) \cap L = \{\} \) and
A-prop: \( (R' \cup \text{all-shared sb}) \cap A = \{\} \) and
R-acq-prop: \( (A' \cup \text{all-acquired sb}) \cap R = \{\} \) and
R-prop: \( (L' \cup \text{all-unshared sb}) \cap R = \{\} \) and
R-prop-sh: \( (R' \cup \text{all-shared sb}) \cap R = \{\} \) and
\[ A'^{-\text{shared-owns}}: A' \subseteq \text{dom } S \cup O \]
\[ L'^{-\text{A}': L'} \subseteq A' \text{ and } A'^{-\text{R}': A'} \cap R = \{\} \]

by (clarsimp simp add: Ghost sb)

from L-prop obtain \( R' \cap L = \{\} \) and acq-L: \( \text{all-shared sb} \cap L = \{\} \)
by blast
from A-prop obtain \( R' \cap A = \{\} \) and acq-A: \( \text{all-shared sb} \cap A = \{\} \)
by blast
from R-acq-prop obtain \( A' \cap R = \{\} \) and acq-R: \( \text{all-acquired sb} \cap R = \{\} \)
by blast
from R-prop obtain \( L' \cap R = \{\} \) and unsh-R: \( \text{all-unshared sb} \cap R = \{\} \)
by blast
from R-prop-sh obtain \( R' \cap R = \{\} \) and sh-R: \( \text{all-shared sb} \cap R = \{\} \)
by blast

from Cons.hyps [OF consis' acq-L acq-A acq-R unsh-R sh-R ] have \( \text{share sb } ((S \oplus W'_R \ominus A' \ L') \oplus W \ominus A \ L) = \text{share sb } ((S \oplus W'_R \ominus A'L') \oplus W \ominus A \ L).\)

moreover
from \( R'^{-L} L'^{-R} R'^{-R} R'^{-A} A'^{-R} \)
have \( ((S \oplus W R \ominus A L) \oplus W R \ominus A L) = ((S \oplus W'_R \ominus A'L') \oplus W \ominus A \ L) \)
apply –
apply (rule ext)
apply (clarsimp simp add: augment-shared-def restrict-shared-def)
apply (auto split: if-split_asm option.splits)
done

ultimately
have \( \text{share sb } ((S \oplus W R \ominus A L) \oplus W R \ominus A L) = \text{share sb } ((S \oplus W'_R \ominus A'L') \oplus W \ominus A \ L) \)

by simp
then
show ?thesis
by (clarsimp simp add: Ghost sb)
qed
qed

lemma share-all-until-volatile-write-commute:
\[ \land S R L. [\text{ownership-distinct ts; sharing-consis } S \text{ ts}; \]
\[ \forall i p \text{ is } O R D \ominus \text{sb}. i < \text{length ts } \rightarrow \text{tsli}=(p, is, \ominus, \text{sb}, D,O,R) \]
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ L = {};
∀ i p is O R D ⊤ sb. i < length ts → ts!i=(p,is,ϑ, sb,D,O, R) →
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ L = {};
∀ i p is O R D ⊤ sb. i < length ts → ts!i=(p,is,ϑ, sb,D,O, R) →
all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {};
∀ i p is O R D ⊤ sb. i < length ts → ts!i=(p,is,ϑ, sb,D,O, R) →
all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {};
∀ i p is O R D ⊤ sb. i < length ts → ts!i=(p,is,ϑ, sb,D,O, R) →
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {}

⇒
share-all-until-volatile-write ts S ⊕ R ⊕_{A} L = share-all-until-volatile-write ts (S ⊕ R ⊕_{A} L)

proof (induct ts)
case Nil
  thus ?case by simp
next
case (Cons t ts)
obtain p is O R D ⊤ sb where
t := (p,is,ϑ, sb,D,O, R)
by (cases t)
have dist: ownership-distinct (t#ts) by fact
then interpret ownership-distinct t#ts.
have consis: sharing-consis S (t#ts) by fact
then interpret sharing-consis S t#ts.

have L-prop: ∀ i p is O R D ⊤ sb. i < length (t#ts) → (t#ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ L = {} by fact
hence L-prop': ∀ i p is O R D ⊤ sb. i < length (ts) → (ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ L = {} by force
have A-prop: ∀ i p is O R D ⊤ sb. i < length (t#ts) → (t#ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ L = {} by force
hence A-prop': ∀ i p is O R D ⊤ sb. i < length (ts) → (ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ L = {} by force
have R-prop-acq: ∀ i p is O R D ⊤ sb. i < length (t#ts) → (t#ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {} by fact
hence R-prop-acq': ∀ i p is O R D ⊤ sb. i < length (ts) → (ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {} by force
have R-prop: ∀ i p is O R D ⊤ sb. i < length (t#ts) → (t#ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {} by fact
hence R-prop': ∀ i p is O R D ⊤ sb. i < length (ts) → (ts)!i=(p,is,ϑ, sb,D,O, R) →
  all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) ∩ R = {}
by force

**have** R-prop-sh: \(\forall i \; p \text{ is } O \; R \; D \; \emptyset \; sb. \; i < \text{length } (t \# ts) \rightarrow (t \# ts)!i = (p, is, \emptyset, sb, D, O, R)\)

\[\rightarrow \]

all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap R = \{\} by fact

**hence** R-prop-sh': \(\forall i \; p \text{ is } O \; R \; D \; \emptyset \; sb. \; i < \text{length } (ts) \rightarrow (ts)!i = (p, is, \emptyset, sb, D, O, R)\)

\[\rightarrow \]

all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap R = \{\} by force

**from** ownership-distinct-tl [OF dist]
**have** dist': ownership-distinct ts.

**from** sharing-consis-tl [OF consis]
**have** consis': sharing-consis \(S\) ts.

then

interpret** consis': sharing-consis \(S\) ts.

**from** L-prop [rule-format, of 0 p is \(\emptyset\) sb \(D\) \(O\)] t
**have** sh-L: all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap L = \{\}

by simp

**from** A-prop [rule-format, of 0 p is \(\emptyset\) sb \(D\) \(O\)] t
**have** sh-A: all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap A = \{\}

by simp

**from** R-prop-acq [rule-format, of 0 p is \(\emptyset\) sb \(D\) \(O\)] t
**have** acq-R: all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap R = \{\}

by simp

**from** R-prop [rule-format, of 0 p is \(\emptyset\) sb \(D\) \(O\)] t
**have** unsh-R: all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap R = \{\}

by simp

**from** R-prop-sh [rule-format, of 0 p is \(\emptyset\) sb \(D\) \(O\)] t
**have** sh-R: all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \cap R = \{\}

by simp

**from** sharing-consis [of 0, simplified, OF t]
**have** sharing-consistent \(S\) \(O\) sb.

**from** sharing-consistent-takeWhile [OF this]
**have** consis-sb: sharing-consistent \(S\) \(O\) (takeWhile (Not o is-volatile-Write_{sb}) sb).

**from** share-commute [OF consis-sb sh-L sh-A acq-R unsh-R sh-R]
**have** share-eq:

\[(\text{share} \text{ (takeWhile (Not o is-volatile-Write}_{sb}) sb) \ (S \oplus_{W} R \ominus_{A} L)) = \]

\[(\text{share} \text{ (takeWhile (Not o is-volatile-Write}_{sb}) sb) \ S) \oplus_{W} R \ominus_{A} L.\]

let \(\mathcal{S}' = (\text{share} \text{ (takeWhile (Not o is-volatile-Write}_{sb}) sb) \ S)\)
from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom ?S' = dom S ⊆ O.
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: dom S − dom ?S' ⊆ all-acquired (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) ∪ O
by simp

have sep:
∀ i < length ts. let (-,-,-,sb'_i,-,-,) = ts! i in
  all-acquired sb'_i ∩ dom S − dom ?S' = {} ∧
  all-unshared sb'_i ∩ dom ?S' − dom S = {}
proof –
{
  fix i p_i is_i R_i D_i θ_i sb_i
  assume i-bound: i < length ts
  assume ts-i: ts ! i = (p_i,is_i,θ_i,sb_i,D_i,R_i)
  have all-acquired sb_i ∩ dom S − dom ?S' = {} ∧
       all-unshared sb_i ∩ dom ?S' − dom S = {}
    proof –
  from ownership-distinct [of 0 Suc i] ts-i t i-bound
    have dist: (O ∪ all-acquired sb) ∩ (O ∪ all-acquired sb_i) = {} 
    by force

  from dist unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-Write_{sb}) sb]
    have all-acquired sb_i ∩ dom S − dom ?S' = {}
      by blast

  moreover

  from sharing-consis [of Suc i] ts-i i-bound
    have sharing-consistent S O_i sb_i 
      by force
  from unshared-acquired-or-owned [OF this]
    have all-unshared sb_i ⊆ all-acquired sb_i ∪ O_i.
  with dist fresh-owned
    have all-unshared sb_i ∩ dom ?S' − dom S = {}
      by blast

  ultimately show ?thesis by simp
  qed
  }
thus ?thesis
  by (fastforce simp add: Let-def)
qed
from consis'.sharing-consis-preservation [OF sep]

322
have sharing-consis': sharing-consis (share (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) S) ts.

from Cons.hyps [OF dist' sharing-consis' L-prop' A-prop' R-prop-acq' R-prop'sh']

have share-all-until-volatile-write ts ?S' ⊕_W R ⊆_A L = 
share-all-until-volatile-write ts (?S' ⊕_W R ⊆_A L).

then

have share-all-until-volatile-write ts 
?S' ⊕_W R ⊆_A L = 
share-all-until-volatile-write ts (?S' ⊕_W R ⊆_A L)
by (simp add: share-eq)
then

show ?case
by (simp add: t)

qed

lemma share-append-Ghost_{sb}:
\[\forall S. \text{outstanding-refs is-volatile-Write}_{sb} \text{ sb } = \{\} \implies (\text{share (sb @ [Ghost}_{sb} A L R W]) S) = (\text{share sb S}) \oplus_W R \ominus_A L\]
apply (induct sb)
apply simp
subgoal for a sb S
apply (case-tac a)
apply auto
done
done

lemma share-append-Ghost_{sb}':
\[\forall S. \text{outstanding-refs is-volatile-Write}_{sb} \text{ sb } \neq \{\} \implies (\text{share (takeWhile (Not ◦ is-volatile-Write}_{sb}) (sb @ [Ghost}_{sb} A L R W]) S) = (\text{share (takeWhile (Not ◦ is-volatile-Write}_{sb}) sb) S)\]
apply (induct sb)
apply simp
subgoal for a sb S
apply (case-tac a)
apply force+
done
done

lemma share-all-until-volatile-write-append-Ghost_{sb}:
assumes no-out-VWrite_{sb}: outstanding-refs is-volatile-Write_{sb} sb = {}
shows \[\forall S i. \text{[ownership-distinct ts; sharing-consis S ts;}
\begin{align*}
&i < \text{length ts;} \quad ts!i = (p, is, \emptyset, sb, D, O, R); \\
&\forall j p \in O \in D \ominus_0 sb. j < \text{length ts } \implies i \neq j \implies ts!j = (p, is, \emptyset, sb, D, O, R) \implies \\
& \text{all-shared (takeWhile (Not ◦ is-volatile-Write}_{sb}) sb) \cap L = \{}; \\
&\forall j p \in O \in D \ominus_0 sb. j < \text{length ts } \implies i \neq j \implies ts!j = (p, is, \emptyset, sb, D, O, R) \implies \\
& \text{all-shared (takeWhile (Not ◦ is-volatile-Write}_{sb}) sb) \cap A = \{};
\end{align*}\]
∀ j p is O R D ⊥ sb. j < length ts → i ≠ j → ts![j]=(p,is,θ,sb,D,O,R) →
all-shared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ R = {}

∀ j p is O R D ⊥ sb. j < length ts → i ≠ j → ts![j]=(p,is,θ,sb,D,O,R) →
all-unshared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ R = {}

∀ j p is O R D ⊥ sb. j < length ts → i ≠ j → ts![j]=(p,is,θ,sb,D,O,R) →
all-shared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ R = {}

⇒
share-all-until-volatile-write (ts[i := (p', is', θ', sb ⊙ [Ghost_{sb} A L R W], D', O')]) S
= share-all-until-volatile-write ts S ⊕_W R ⊕_A L

proof (induct ts)
case Nil
thus ?case by simp
next
case (Cons t ts)
obtain p_t is_t O_t R_t D_t acq_t ⊥_t sb_t where
t:: t=(p_t,is_t,θ_t,acq_t,⊥_t,ts)
by (cases t)
have dist: ownership-distinct (t#ts) by fact
then interpret ownership-distinct t#ts.
have consis: sharing-consis S (t#ts) by fact
then interpret sharing-consis S t#ts.

have L-prop: ∀ j p is O R D ⊥ sb. j < length (t#ts) → i ≠ j →
(t#ts)[j]=(p,is,θ,sb,D,O,R) →
all-shared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ L = {} by fact

have A-prop: ∀ j p is O R D ⊥ sb. j < length (t#ts) → i ≠ j →
(t#ts)[j]=(p,is,θ,sb,D,O,R) →
all-shared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ A = {} by fact

have R-prop-acq: ∀ j p is O R D ⊥ sb. j < length (t#ts) → i ≠ j→
(t#ts)[j]=(p,is,θ,sb,D,O,R) →
all-acquired (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ R = {} by fact

have R-prop: ∀ j p is O R D ⊥ sb. j < length (t#ts) → i ≠ j→
(t#ts)[j]=(p,is,θ,sb,D,O,R) →
all-unshared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ R = {} by fact

have R-prop-sh: ∀ j p is O R D ⊥ sb. j < length (t#ts) → i ≠ j→
(t#ts)[j]=(p,is,θ,sb,D,O,R) →
all-shared (takeWhile (Not ° is-volatile-Write_{sb}) sb) ∩ R = {} by fact

from ownership-distinct-tl [OF dist]
have dist': ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis': sharing-consis S ts.
then interpret consis': sharing-consis S ts.

324
from sharing-consis [of 0, simplified, OF t]

have sharing-consistent \( S \ O_I \ sb_t \).

from sharing-consistent-takeWhile [OF this]

have consis-sb: sharing-consistent \( S \ O_I \) (takeWhile \((\not \circ \text{is-volatile-Write}_{sb})\) sb_t).

let \( ?S' = (\text{share} \ (\text{takeWhile} \ ((\not \circ \text{is-volatile-Write}_{sb})\) sb_t) \ S) \)

from freshly-shared-owned [OF consis-sb]

have fresh-owned: dom \( ?S' - \text{dom} \ S \subseteq O_t \).

from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]

have unshared-acq-owned: dom \( S - \text{dom} \ ?S' \subseteq \text{all-acquired} \ (\text{takeWhile} \ ((\not \circ \text{is-volatile-Write}_{sb})\) sb_t) \cup O_t \)

by simp

have sep:
\[ \forall i < \text{length } ts. \text{let } (\_,\_,\_,\_,\_,\_,\_,sb_i) = ts!i \text{ in} \]
\[ \text{all-acquired } sb_i \cap \text{dom} \ S - \text{dom} \ ?S' = \{\} \land \]
\[ \text{all-unshared } sb_i \cap \text{dom} \ ?S' - \text{dom} \ S = \{\} \]

proof -
\{ 
  \fix i \text{ p_i } is_i \ R_i \ D_i \ acq_i \ \emptyset_i \ \text{sb_i} \\
  \assume i-bound: i < \text{length } ts \\
  \assume ts-i: \text{ts} ! i = (p_i, is_i, \emptyset_i, sb_i, D_i, O_i, R_i) \\
  \have all-acquired sb_i \cap \text{dom} \ S - \text{dom} \ ?S' = \{\} \land \\
  \text{all-unshared } sb_i \cap \text{dom} \ ?S' - \text{dom} \ S = \{\} \\
  \proof - \\
  \from ownership-distinct [of 0 Suc i] ts-i t i-bound \\
  \have \text{dist: } (O_I \cup \text{all-acquired } sb_i) \cap (O_I \cup \text{all-acquired } sb_i) = \{\} \\
  \by force \\
  \from dist unshared-acq-owned all-acquired-takeWhile [of (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb_i}] \\
  \have \text{all-acquired } sb_i \cap \text{dom} \ S - \text{dom} \ ?S' = \{\} \\
  \by blast \\
  \moreover \\
  \from sharing-consis [of Suc i] ts-i t i-bound \\
  \have sharing-consistent \( S \ O_I \ sb_i \) \\
  \by force \\
  \from unshared-acquired-or-owned [OF this] \\
  \have \text{all-unshared } sb_i \subseteq \text{all-acquired } sb_i \cup O_I, \\
  \with dist fresh-owned \\
  \have \text{all-unshared } sb_i \cap \text{dom} \ ?S' - \text{dom} \ S = \{\} \\
  \by blast \\
  \ultimately show ?\text{thesis by simp} \\
  \qed \\
\}

325
thus thesis
  by (fastforce simp add: Let-def)
qed

from consis\'.sharing-consis-preservation [OF sep]
have sharing-consis\': sharing-consis (share (takeWhile (Not o is-volatile-Write_{sb} \, sb) \, S) \, S) \, ts.

show ?case
proof (cases i)
  case 0
  with t Cons.prems have eqs: p_\, t = p \, i s_\, t = is_\, t = is_\, O_\, t = O_\, R_\, t = R_\, \theta_\, t = \theta_\, sb_\, t = sb_\, D_\, t = D
    by auto
  from no-out-VWrite_{sb}
  have flush-all: takeWhile (Not o is-volatile-Write_{sb} \, sb) \, sb = sb
    by (auto simp add: outstanding-refs-conv)
  have flush-all': takeWhile (Not o is-volatile-Write_{sb} \, sb@[Ghost_{sb} A L R W]) = sb@[Ghost_{sb} A L R W]
    by (auto simp add: outstanding-refs-conv)
  have share-eq:
    (share (takeWhile (Not o is-volatile-Write_{sb} \, sb@[Ghost_{sb} A L R W])) \, S) =
      (share (takeWhile (Not o is-volatile-Write_{sb} \, sb) \, S) \oplus_W R \ominus_A L
    apply (simp only: flush-all flush-all')
    apply (rule share-append-Ghost_{sb} [OF no-out-VWrite_{sb}])
    done
  from L-prop 0 have L-prop':
    \forall \, i \, p \, is \, O \, R \, D \cap \, sb.
    i < \, length \, ts \longrightarrow
    ts \mid i = (p, \, is, \, \theta, \, sb, \, D, \, O, R) \longrightarrow
    all-shared (takeWhile (Not o is-volatile-Write_{sb} \, sb) \, \cap \, L = \{\})
    apply clarsimp
    subgoal for i1 \, p \, is \, O \, R \, D \cap \, sb
      apply (drule-tac x=Suc i1 in spec)
      apply auto
      done
    done
  from A-prop 0 have A-prop':
    \forall \, i \, p \, is \, O \, R \, D \cap \, sb.
    i < \, length \, ts \longrightarrow
    ts \mid i = (p, \, is, \, \theta, \, sb, \, D, \, O, R) \longrightarrow
    all-shared (takeWhile (Not o is-volatile-Write_{sb} \, sb) \, \cap \, A = \{\})
    apply clarsimp
    subgoal for i1 \, p \, is \, O \, R \, D \cap \, sb
      apply (fastforce simp add: Let-def)
apply (drule-tac x=Suc i1 in spec)
apply auto
done
done
from R-prop-acq 0 have R-prop-acq':
\forall i p is \sigma \rho D \theta sb. \ i < \text{length } ts \implies ts!i=(p,\text{is},\theta,\text{sb},D,\sigma,\rho) \implies
\text{all-acquired } \left(\text{takeWhile } \left(\text{Not } \circ \text{is-volatile-Write}_{sb}\right) \text{ sb}\right) \cap R = \{\}
apply clarsimp
subgoal for i1 p is \sigma \rho D \theta sb
apply (drule-tac x=Suc i1 in spec)
apply auto
done
done
from R-prop 0 have R-prop':
\forall i p is \sigma \rho D \theta sb. \ i < \text{length } ts \implies ts!i=(p,\text{is},\theta,\text{sb},D,\sigma,\rho) \implies
\text{all-unshared } \left(\text{takeWhile } \left(\text{Not } \circ \text{is-volatile-Write}_{sb}\right) \text{ sb}\right) \cap R = \{\}
apply clarsimp
subgoal for i1 p is \sigma \rho D \theta sb
apply (drule-tac x=Suc i1 in spec)
apply auto
done
done
from R-prop-sh 0 have R-prop-sh':
\forall i p is \sigma \rho D \theta sb. \ i < \text{length } ts \implies ts!i=(p,\text{is},\theta,\text{sb},D,\sigma,\rho) \implies
\text{all-shared } \left(\text{takeWhile } \left(\text{Not } \circ \text{is-volatile-Write}_{sb}\right) \text{ sb}\right) \cap R = \{\}
apply clarsimp
subgoal for i1 p is \sigma \rho D \theta sb
apply (drule-tac x=Suc i1 in spec)
apply auto
done
done
from share-all-until-volatile-write-commute [OF dist' sharing-consis' L-prop' A-prop'
R-prop-acq' R-prop'
R-prop-sh']

have share-all-until-volatile-write ts (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}) S \oplus_{W} R \oplus_{A} L) =
share-all-until-volatile-write ts (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}) S) \oplus_{W} R \oplus_{A} L
by (simp add: eqs)
with share-eq
show ?thesis
by (clarsimp simp add: 0 t)
next
case (Suc k)
from L-prop Suc
have \textbf{L-prop}': $\forall j \ p \in \text{ORD} \ θ \ sb. \ j < \text{length} (ts) \rightarrow k \neq j \rightarrow (ts)[j]=(p,\text{is},\text{sb},D,O,R) \rightarrow$

\hspace*{2cm} all-shared \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap L = \{\} \text{ \ by force}

from \textbf{A-prop Succ}

have \textbf{A-prop}': $\forall j \ p \in \text{ORD} \ θ \ sb. \ j < \text{length} (ts) \rightarrow k \neq j \rightarrow (ts)[j]=(p,\text{is},\text{sb},D,O,R) \rightarrow$

\hspace*{2cm} all-shared \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap \text{A} = \{\} \text{ \ by force}

from \textbf{R-prop-acq Succ} have \textbf{R-prop-acq}' :

\hspace*{1cm} $\forall j \ p \in \text{ORD} \ θ \ sb. \ j < \text{length ts} \rightarrow k \neq j \rightarrow \text{ts}[j]=(p,\text{is},\text{sb},D,O,R) \rightarrow$

\hspace*{2cm} all-acquired \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap \text{R} = \{\} \text{ \ by force}

from \textbf{R-prop Succ} have \textbf{R-prop}':

\hspace*{1cm} $\forall j \ p \in \text{ORD} \ θ \ sb. \ j < \text{length ts} \rightarrow k \neq j \rightarrow \text{ts}[j]=(p,\text{is},\text{sb},D,O,R) \rightarrow$

\hspace*{2cm} all-unshared \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap \text{R} = \{\} \text{ \ by force}

from \textbf{R-prop-sh Succ} have \textbf{R-prop-sh}':

\hspace*{1cm} $\forall j \ p \in \text{ORD} \ θ \ sb. \ j < \text{length ts} \rightarrow k \neq j \rightarrow \text{ts}[j]=(p,\text{is},\text{sb},D,O,R) \rightarrow$

\hspace*{2cm} all-shared \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap \text{R} = \{\} \text{ \ by force}

from \textbf{Cons.prems Succ} obtain \textbf{k-bound}: $k < \text{length ts} \text{ \ and } \text{ts-k}: \text{ts}[k] = (p, \text{is}, \text{sb}, D, O, R)$

by auto

from \textbf{Cons.hyps \ [OF dist \textbf{'} sharing-consis \textbf{'} \textbf{k-bound} \textbf{'} ts-k \textbf{'} \textbf{L-prop} \textbf{'} \textbf{A-prop} \textbf{'} \textbf{R-prop-acq} \textbf{' R-prop'} \textbf{'} \textbf{R-prop-sh} \textbf{' ]} show \ ?thesis

by (clarsimp simp add: t Suc)

qed

lemma \textbf{share-domain-changes}:

\hspace*{1cm} $\forall S' \ a \in \text{all-shared sb} \cup \text{all-unshared sb} \Longrightarrow \text{share sb S'} a = \text{share sb S a}$

proof (induct sb)

\hspace*{2cm} case Nil \textbf{thus} ?case by simp

next

\hspace*{2cm} case (Cons x sb)

\hspace*{2cm} show ?case

proof (cases x)

\hspace*{3cm} case (Write_{sb} \text{ volatile a'} sop v A L R W)

\hspace*{3cm} show \ ?thesis

proof (cases volatile)

\hspace*{4cm} case True

\hspace*{4cm} note volatile=this
from Cons.prems obtain a-in: a ∈ R ∪ all-shared sb ∪ L ∪ all-unshared sb
by (clarsimp simp add: Write\_sb True)
show ?thesis
proof (cases a ∈ R)
case True
from True have (S' ⊕ W R ⊋ A L) a = (S ⊕ W R ⊋ A L) a
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq [where $S' = S' ⊕ W R ⊋ A L$ and $S = S ⊕ W R ⊋ A L$, OF this]
have share sb (S' ⊕ W R ⊋ A L) a = share sb (S ⊕ W R ⊋ A L) a
by auto
then show ?thesis
by (clarsimp simp add: Write\_sb volatile)
next
case False
note not-R = this
show ?thesis
proof (cases a ∈ L)
case True
from not-R True have (S' ⊕ W R ⊋ A L) a = (S ⊕ W R ⊋ A L) a
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq [where $S' = S' ⊕ W R ⊋ A L$ and $S = S ⊕ W R ⊋ A L$, OF this]
have share sb (S' ⊕ W R ⊋ A L) a = share sb (S ⊕ W R ⊋ A L) a
by auto
then show ?thesis
by (clarsimp simp add: Write\_sb volatile)
next
case False
with Cons a-in have a ∈ all-shared sb ∪ all-unshared sb
by auto
from Cons.hyps [OF this]
show ?thesis by (clarsimp simp add: Write\_sb volatile)
qed
qued
next
case False with Cons show ?thesis by (auto simp add: Write\_sb)
qed
next
case Read\_sb with Cons show ?thesis by (auto)
next
case Prog\_sb with Cons show ?thesis by (auto)
next
case (Ghost\_sb A L R W)
from Cons.prems obtain a-in: a ∈ R ∪ all-shared sb ∪ L ∪ all-unshared sb
by (clarsimp simp add: Ghost\_sb)
show ?thesis
proof (cases a ∈ R)
case True
from True have (S' ⊕ W R ⊋ A L) a = (S ⊕ W R ⊋ A L) a
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq \where S' = S' \oplus_W R \ominus_A L and S = S \oplus_W R \ominus_A L, OF this
have share sb (S' \oplus W R \ominus A L) a = share sb (S \oplus W R \ominus A L) a
  by auto
then show \?thesis
  by (clarsimp simp add: Ghost_{sb})

next
case False
note not-R = this
show \?thesis
proof (cases a \in L)
case True
from not-R True have (S' \oplus W R \ominus A L) a = (S \oplus W R \ominus A L) a
  by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq \where S' = S' \oplus_W R \ominus_A L and S = S \oplus_W R \ominus_A L, OF this
have share sb (S' \oplus_W R \ominus_A L) a = share sb (S \oplus_W R \ominus_A L) a
  by auto
then show \?thesis
  by (clarsimp simp add: Ghost_{sb})
next
case False
with not-R a-in have a \in all-shared sb \cup all-unshared sb
  by auto
from Cons.hyps [OF this]
show \?thesis by (clarsimp simp add: Ghost_{sb})
qed
qed
qed

lemma share-domain-changesX:
  \all S \in S'. \forall a \in X. S' a = S a
  \implies a \in all-shared sb \cup all-unshared sb \cup X \implies share sb S' a = share sb S a
proof (induct sb)
case Nil thus \?case by simp
next
case (Cons x sb)
then have shared-eq: \forall a \in X. S' a = S a
  by auto
show \?case
proof (cases x)
case (Write_{sb} volatile a' sop v A L R W)
show \?thesis
proof (cases volatile)
case True
  note volatile=this
from Cons.prems obtain a-in: a \in R \cup all-shared sb \cup L \cup all-unshared sb \cup X
  by (clarsimp simp add: Write_{sb} True)
show \?thesis
proof (cases a \in R)
  case True
from True have \((S' \oplus W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a\)
  by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq [where \(S' = S' \oplus W R \ominus_A L\) and \(S = S \oplus_W R \ominus_A L\), OF this]
have share sb \((S' \oplus_W R \ominus_A L) a = \) share sb \((S \oplus_W R \ominus_A L) a\)
  by auto
then show ?thesis
  by (clarsimp simp add: Write sb volatile)

next
case False
note not-R = this
show ?thesis
proof (cases a \(\in L\))
case True
  from not-R True have \((S' \oplus_W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a\)
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq [where \(S' = S' \oplus W R \ominus_A L\) and \(S = S \oplus_W R \ominus_A L\), OF this]
  have share sb \((S' \oplus_W R \ominus_A L) a = \) share sb \((S \oplus_W R \ominus_A L) a\)
    by auto
  then show ?thesis
    by (clarsimp simp add: Write sb volatile)
next
case False
  from shared-eq have shared-eq' \(\forall a \in X. (S' \oplus_W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a\)
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  from False not-R a-in have \(a \in\) all-shared sb \(\cup\) all-unshared sb \(\cup\) X
    by auto
  from Cons.hyps [OF shared-eq' this]
  show ?thesis
    by (clarsimp simp add: Write sb volatile)
qed
qed
next
case False with Cons show ?thesis
  by (auto simp add: Write sb)
qed
next
case Read sb with Cons show ?thesis
  by (auto)
next
case Prog sb with Cons show ?thesis
  by (auto)
next
case (Ghost sb A L R W)
from Cons.prems obtain a-in: \(a \in\) R \(\cup\) all-shared sb \(\cup\) L \(\cup\) all-unshared sb \(\cup\) X
  by (clarsimp simp add: Ghost sb)
show ?thesis
proof (cases a \(\in\) R)
case True
  from True have \((S' \oplus_W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a\)
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  from share-shared-eq [where \(S' = S' \oplus W R \ominus_A L\) and \(S = S \oplus_W R \ominus_A L\), OF this]
    have share sb \((S' \oplus_W R \ominus_A L) a = \) share sb \((S \oplus_W R \ominus_A L) a\)
      by auto

qed
331
then show ?thesis
  by (clarsimp simp add: Ghost sb)
next
case False
note not-R = this
show ?thesis
proof (cases a ∈ L)
case True
  from not-R True have \((S' ⊕ W R ⊕ A L) a = (S ⊕ W R ⊕ A L) a\)
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  from share-shared-eq [where \(S' = S' ⊕ W R ⊕ A L\) and \(S = S ⊕ W R ⊕ A L\), OF this]
  have share sb \((S' ⊕ W R ⊕ A L) a = \text{share sb } (S ⊕ W R ⊕ A L) a\)
    by auto
  then show ?thesis
    by (clarsimp simp add: Ghost sb)
next
case False
  from shared-eq have \(\forall a ∈ X. (S' ⊕ W R ⊕ A L) a = (S ⊕ W R ⊕ A L) a\)
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  from False not-R a-in have \(a ∈ \text{all-shared sb} \cup \text{all-unshared sb} \cup X\)
    by auto
  from Cons.hyps [OF shared-eq' this]
  show ?thesis by (clarsimp simp add: Ghost sb)
qed
qed
qed
lemma share-unchanged:
\(\bigwedge S. a \notin \text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb} \implies \text{share sb } S a = S a\)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a' s o p v A L R W)
show ?thesis
proof (cases volatile)
case True
  note volatile=this
  from Cons.prems obtain a-R: a \notin R and a-L: a \notin L and a-A: a \notin A
    and a': a \notin all-shared sb \cup all-unshared sb \cup all-acquired sb
    by (clarsimp simp add: Write sb True)
  from Cons.hyps [OF a']
  have share sb \((S ⊕ W R ⊕ A L) a = (S ⊕ W R ⊕ A L) a\).
  moreover
  from a-R a-L a-A have \((S ⊕ W R ⊕ A L) a = S a\)
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  ultimately
show ?thesis
  by (clarsimp simp add: Write sb True)
next
  case False with Cons
  show ?thesis by (auto simp add: Write sb)
next
  case Read sb with Cons
  show ?thesis by (auto)
next
  case Prog sb with Cons
  show ?thesis by (auto)
next
  case (Ghost sb A L R W)
  from Cons.prems obtain a-R: a /∈ R and a-L: a /∈ L and a-A: a /∈ A
  and a': a /∈ all-shared sb ∪ all-unshared sb ∪ all-acquired sb
  by (clarsimp simp add: Ghost sb)
  from Cons.hyps [OF a'] have share sb (S⊕W R ⊖A L) a = (S⊕W R ⊖A L) a .
  moreover
  from a-R a-L a-A have (S⊕W R ⊖A L) a = S a
  by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  ultimately
  show ?thesis
  by (clarsimp simp add: Ghost sb)
qed

lemma share-augment-release-commute:
assumes dist: (R ∪ L ∪ A) ∩ (all-shared sb ∪ all-unshared sb ∪ all-acquired sb) = {}
sows (share sb (S⊕W R ⊖A L) = share sb (S⊕W R ⊖A L)
proof −
  from dist have shared-eq: ∀ a ∈ all-acquired sb. (S⊕W R ⊖A L) a = S a
  by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
  { fix a
    assume a-in: a ∈ all-shared sb ∪ all-unshared sb ∪ all-acquired sb
    from share-domain-changesX [OF shared-eq this]
    have share sb (S⊕W R ⊖A L) a = share sb S a.
    also
    from dist a-in have ... = (share sb (S⊕W R ⊖A L) a
    by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
    finally have share sb (S⊕W R ⊖A L) a = (share sb S⊕W R ⊖A L) a.
  }
  moreover
  { fix a
    assume a-notin: a /∈ all-shared sb ∪ all-unshared sb ∪ all-acquired sb
    from share-unchanged [OF a-notin]
    have share sb (S⊕W R ⊖A L) a = (S⊕W R ⊖A L) a.
    moreover
    from share-unchanged [OF a-notin]

333
have \( \text{share sb } S \ a = S \ a \).
hence \((\text{share sb } S \oplus_\mathsf{W} R \ominus_\mathsf{A} L) \ a = (S \oplus_\mathsf{W} R \ominus_\mathsf{A} L) \ a \)
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
ultimately have \( \text{share sb } (S \oplus_\mathsf{W} R \ominus_\mathsf{A} L) \ a = (\text{share sb } S \oplus_\mathsf{W} R \ominus_\mathsf{A} L) \ a \)
by simp
\}
ultimately show \(?\text{thesis} \)
apply –
apply (rule ext)
subgoal for \(x \)
apply (case-tac \(x \in \text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb} \))
apply auto
done

done
qed

lemma \(\text{share-append-commute:} \)
\(\forall \text{ys } S. \ (\text{all-shared xs} \cup \text{all-unshared xs} \cup \text{all-acquired xs}) \cap \)
\((\text{all-shared ys} \cup \text{all-unshared ys} \cup \text{all-acquired ys}) = \{\} \)
\(\implies \text{share xs } (\text{share \text{ys } S}) = \text{share \text{ys } (\text{share } \text{xs } S}) \)
proof (induct \(xs \))
case Nil thus \(?\text{case} \) by simp
next
case (Cons \(x \) \(xs \))
show \(?\text{case} \)
proof (cases \(x \))
case (Write \(\text{sb} \) volatile a sop v A L R W)
show \(?\text{thesis} \)
proof (cases volatile)
case True
note volatile=this
from Cons.prems have
\(\text{dist}' (\text{all-shared xs} \cup \text{all-unshared xs} \cup \text{all-acquired xs}) \cap \)
\((\text{all-shared ys} \cup \text{all-unshared ys} \cup \text{all-acquired ys}) = \{\} \)
apply (clarsimp simp add: Write \(\text{sb} \) True)
apply blast
done
from Cons.prems have
\(\text{dist: } (R \cup L \cup A) \cap (\text{all-shared ys} \cup \text{all-unshared ys} \cup \text{all-acquired ys}) = \{\} \)
apply (clarsimp simp add: Write \(\text{sb} \) True)
apply blast
done
from \(\text{share-augment-release-commute} \ [\text{OF dist}] \)
have \(\text{share \text{ys } S} \oplus_\mathsf{W} R \ominus_\mathsf{A} L) = \text{share \text{ys } (S} \oplus_\mathsf{W} R \ominus_\mathsf{A} L). \)
\(\text{with Cons.hyps } [\text{OF dist}] \]
show \(?\text{thesis} \)
by (clarsimp simp add: Write \(\text{sb} \) True)
next
case False with Cons show \(?\text{thesis} \)
by (clarsimp simp add: Write sb False)
qed

next

\textbf{case} Read\_{sb} \textbf{with} Cons \textbf{show} \ \textbf{?thesis} \ \textbf{by} \ \textbf{auto}

next

\textbf{case} Prog_{sb} \textbf{with} Cons \textbf{show} \ \textbf{?thesis} \ \textbf{by} \ \textbf{auto}

next

\textbf{case} (Ghost_{sb} A L R W)

\textbf{from} Cons.prems \textbf{have}

\textbf{dist}': \ (all-shared xs \cup all-unshared xs \cup all-acquired xs) \cap

\quad (all-shared ys \cup all-unshared ys \cup all-acquired ys) = \{\}

\textbf{apply} (clarsimp simp add: Ghost sb)

\textbf{apply} blast

done

\textbf{from} Cons.prems \textbf{have}

\textbf{dist}: \ (R \cup L \cup A) \cap (all-shared ys \cup all-unshared ys \cup all-acquired ys) = \{\}

\textbf{apply} (clarsimp simp add: Ghost sb)

\textbf{apply} blast

done

\textbf{from} share-augment-release-commute \ [OF dist]

\textbf{have} \share{ys@xs}{S} = \share{xs@ys}{S}.

\textbf{with} Cons.hyps \ [OF dist']

\textbf{show} \ \textbf{?thesis}

\textbf{by} (clarsimp simp add: Ghost sb)
qed

lemma \textbf{share-append-commute}':

\textbf{assumes} \textbf{dist}: \ (all-shared xs \cup all-unshared xs \cup all-acquired xs) \cap

\quad (all-shared ys \cup all-unshared ys \cup all-acquired ys) = \{\}

\textbf{shows} \share{ys@xs}{S} = \share{xs@ys}{S}

\textbf{proof} –

\textbf{from} share-append-commute \ [OF dist] share-append \ [of xs ys] share-append \ [of ys xs]

\textbf{show} \ \textbf{?thesis}

\textbf{by} simp

qed

lemma \textbf{share-all-until-volatile-write-share-commute}:

\textbf{shows} \ \forall S \ (sb'::'a memref list). \ [(ownership-distinct ts; \ sharing-consis S ts; \ \forall i p is O R D \emptyset (sb::'a memref list). \ i < \ length ts

\quad \to \ tshi=(p,is,\emptyset,\emptyset,sb,D,O,R) \to

\quad \ (all-shared (takeWhile (Not o is-volatile-Write sb) sb) \cup

\quad all-unshared (takeWhile (Not o is-volatile-Write sb) sb) \cup

\quad all-acquired (takeWhile (Not o is-volatile-Write sb) sb)) \cap

\quad (all-shared sb' \cup all-unshared sb' \cup all-acquired sb') = \{\}]

\Rightarrow

\textbf{share-all-until-volatile-write ts} (share \sb' S) = \share{sb'}{S}

\textbf{proof} (induct ts)
case Nil
thus \( ?\text{case by simp} \)
next
case (Cons t ts)
obtain \( p_t \) \( s_t \) \( \mathcal{O}_t \) \( D_t \) \( \emptyset_t \) \( s_b_t \) where
\( t = (p_t, s_t, \emptyset_t, s_b_t, D_t, \mathcal{O}_t, \mathcal{R}_t) \)
by (cases t)

let \( ?\text{take} = (\text{takeWhile (Not o is-volatile-Write}_{sb}) s_b_t) \)
have dist: ownership-distinct \( (t#ts) \) by fact
then interpret ownership-distinct \( t#ts \).
have consis: sharing-consis \( S \) \( (t#ts) \) by fact
then interpret sharing-consis \( S \) \( t#ts \).

have dist-prop: \( \forall i \text{ p is } \mathcal{O} \mathcal{R} D \emptyset \text{ sb. i < length (t#ts)} \)
\( \rightarrow (t#ts)!i=(p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}) \rightarrow \)
(\( \text{all-shared (takeWhile (Not o is-volatile-Write}_{sb}) s_b) \) \( \cup \)
\( \text{all-unshared (takeWhile (Not o is-volatile-Write}_{sb}) s_b) \) \( \cup \)
\( \text{all-acquired (takeWhile (Not o is-volatile-Write}_{sb}) s_b) \) \( \cap \)
(\( \text{all-shared sb'} \cup \text{all-unshared sb'} \cup \text{all-acquired sb'} \) \( \cap \)
\( \text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb} \) \( \) \( = \{\} \) by fact
from dist-prop [rule-format, of 0] t
have dist-t: (\( \text{all-shared ?take } \cup \text{all-unshared ?take } \cup \text{all-acquired ?take} \) \( \cap \)
(\( \text{all-shared sb'} \cup \text{all-unshared sb'} \cup \text{all-acquired sb'} \) \( = \{\} \)
apply clarsimp
done
from dist-prop have
dist-prop': \( \forall i \text{ p is } \mathcal{O} \mathcal{R} D \emptyset \text{ sb. i < length ts} \)
\( \rightarrow t\text{tsli}=(p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}) \rightarrow \)
(\( \text{all-shared (takeWhile (Not o is-volatile-Write}_{sb}) s_b) \) \( \cup \)
\( \text{all-unshared (takeWhile (Not o is-volatile-Write}_{sb}) s_b) \) \( \cup \)
\( \text{all-acquired (takeWhile (Not o is-volatile-Write}_{sb}) s_b) \) \( \cap \)
(\( \text{all-shared sb'} \cup \text{all-unshared sb'} \cup \text{all-acquired sb'} \) \( = \{\} \)
apply (clarsimp)
subgoal for \( i \text{ p is } \mathcal{O} \mathcal{R} D \emptyset \text{ sb} \)
apply (drule-tac x=Suc i in spec)
apply clarsimp
done
done
done

from ownership-distinct-tl [OF dist]
have dist': ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis': sharing-consis \( S \) ts.
then interpret consis': sharing-consis \( S \) ts.

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent \( S \) \( \mathcal{O}_t \) \( s_b_t \).
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent $S \cup \mathcal{O}_t$ ?take.

let $?S' = (share ?take S)$

from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom $?S' - dom S \subseteq \mathcal{O}_t$.
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: dom $S - dom ?S'$ \subseteq all-acquired $(\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb_t) \cup \mathcal{O}_t$
by simp

have sep:
\forall i < \text{length ts}. let $(\cdot, \cdot, \cdot, sb', \cdot, \cdot, \cdot)$ = ts!i in
all-acquired $sb' \cap$ dom $S - dom ?S' = \{\}$ \land
all-unshared $sb' \cap$ dom $?S' - dom S = \{\}$
proof (\text{-})
  \{ 
  fix $i$ p$i$ is$i$ $\mathcal{O}_i$ $\mathcal{R}_i$ $D_i$ acqi $\theta_i$ sb$i$
  assume i-bound: $i < \text{length ts}$
  assume ts-i: ts ! i = (p$i$, is$i$, $\theta_i$, sb$i$, $D_i$, $\mathcal{O}_i$, $\mathcal{R}_i$)
  have all-acquired $sb_i \cap$ dom $S - dom ?S' = \{\}$ \land
  all-unshared $sb_i \cap$ dom $?S' - dom S = \{\}$
  proof (\text{-})
  from ownership-distinct [of 0 Suc i] ts-i t i-bound
  have dist: $(\mathcal{O}_t \cup \text{all-acquired sb}_t) \cap (\mathcal{O}_t \cup \text{all-acquired sb}_i) = \{\}$
  by force

  from dist unshared-acq-owned all-acquired-takeWhile [of (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb_t]
  have all-acquired $sb_i \cap$ dom $S - dom ?S' = \{\}$
  by blast
moreover
from sharing-consis [of Suc i] ts-i i-bound
have sharing-consistent $S \cup \mathcal{O}_t$ sb$i$
by force
from unshared-acquired-or-owned [OF this]
have all-unshared sb$i$ \subseteq all-acquired sb$i$ \cup $\mathcal{O}_t$.
with dist fresh-owned
have all-unshared sb$i$ \cap dom $?S' - dom S = \{\}$
by blast
ultimately show \(\text{thesis}\) by simp
qed

thus \(\text{thesis}\)
  by (fastforce simp add: Let-def)

337
from consis’.sharing-consis-preservation [OF sep]

have sharing-consis’: sharing-consis ?S’ ts.

have share-all-until-volatile-write ts (share ?take (share sb’ ?S)) =
    share sb’ (share-all-until-volatile-write ts (share ?take ?S))

proof –
  from share-append-commute [OF dist-t]
  have (share ?take (share sb’ ?S)) = (share sb’ (share ?take ?S)) .
  then
  have share-all-until-volatile-write ts (share ?take (share sb’ ?S)) =
    share-all-until-volatile-write ts (share sb’ (share ?take ?S))
    by (simp)
  also
  from Cons.hyps [OF dist’ sharing-consis’ dist-prop’]
  have ... = share sb’ (share-all-until-volatile-write ts (share ?take ?S)).
  finally show ?thesis .

qed
then show ?case
  by (clarsimp simp add: t)

qed


lemma all-shared-takeWhile-subset: all-shared (takeWhile P sb) ⊆ all-shared sb
using all-shared-append [of (takeWhile P sb) (dropWhile P sb)]
  by auto

lemma all-shared-dropWhile-subset: all-shared (dropWhile P sb) ⊆ all-shared sb
using all-shared-append [of (takeWhile P sb) (dropWhile P sb)]
  by auto

lemma all-unshared-takeWhile-subset: all-unshared (takeWhile P sb) ⊆ all-unshared sb
using all-unshared-append [of (takeWhile P sb) (dropWhile P sb)]
  by auto

lemma all-unshared-dropWhile-subset: all-unshared (dropWhile P sb) ⊆ all-unshared sb
using all-unshared-append [of (takeWhile P sb) (dropWhile P sb)]
  by auto

lemma all-acquired-takeWhile-subset: all-acquired (takeWhile P sb) ⊆ all-acquired sb
using all-acquired-append [of (takeWhile P sb) (dropWhile P sb)]
  by auto

lemma all-acquired-dropWhile-subset: all-acquired (dropWhile P sb) ⊆ all-acquired sb
using all-acquired-append [of (takeWhile P sb) (dropWhile P sb)]
  by auto

lemma share-all-until-volatile-write-flush-commute:
assumes takeWhile-empty: (takeWhile (Not ◦ is-volatile-Write sb) sb) = []
shows ∃ S R L W A i. [ownership-distinct ts; sharing-consis S ts; i < length ts;
    ts!i = (p,is,0,sb,D,O,R)];
∀ i p is \( O \ R \ D \not\emptyset (sb:\text{a memref list}) \), \( i < \text{length ts} \)

\[
\rightarrow ts!i = (p,\text{is,}0,sb,D,O,R) \\
\quad (\text{all-shared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-unshared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-acquired (takeWhile (Not o is-volatile-Write_{sb}O) sb)}) \cap \text{all-shared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-unshared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-acquired (takeWhile (Not o is-volatile-Write_{sb}O) sb)}) = \{\};
\]

∀ j p is \( O \ R \ D \not\emptyset (sb:\text{a memref list}) \), \( j < \text{length ts} \rightarrow i \neq j \rightarrow ts!j = (p,\text{is,}0,sb,D,O,R) \rightarrow \)

\[
\quad (\text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb}) \cap (R \cup L \cup A) = \{\}
\]

\[ \Rightarrow \text{share-all-until-volatile-write (ts[i :=(p',\text{is,}0,sb',D',O',R'])]} (S \oplus W R \odot_{A} L) = \text{share (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \quad \text{(share-all-until-volatile-write ts S \oplus W R \odot_{A} L)}
\]

\[ \text{proof (induct ts)} \]

\[ \text{case (Nil)} \]

\[ \text{thus ?case by simp} \]

\[ \text{next} \]

\[ \text{case (Cons t ts)} \]

\[ \text{obtain} \ p_{t} \text{ is } O_{t} \ D_{t} \ \text{ where} \]

\[ t : t = (p_{t},\text{is,}0,sb_{t},D_{t},O_{t},R_{t}) \]

\[ \text{by (cases t)} \]

\[ \text{let ?take = (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \]

\[ \text{let ?take-sb' = (takeWhile (Not o is-volatile-Write_{sb}O) sb') } \]

\[ \text{let ?drop = (dropWhile (Not o is-volatile-Write_{sb}O) sb)} \]

\[ \text{have dist: ownership-distinct (t#ts) by fact} \]

\[ \text{then interpret ownership-distinct t#ts} . \]

\[ \text{have consis: sharing-consis S (t#ts) by fact} \]

\[ \text{then interpret sharing-consis S t#ts} . \]

\[ \text{have dist-prop: } \forall \ i \ p \ is \ O \ R \ D \not\emptyset \ sb. \ i < \text{length (t#ts)} \]

\[ \rightarrow (t#ts)!i = (p,\text{is,}0,sb,D,O,R) \rightarrow \]

\[ \quad (\text{all-shared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-unshared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-acquired (takeWhile (Not o is-volatile-Write_{sb}O) sb)}) \cap \text{all-shared ?take-sb' \cup all-unshared ?take-sb' \cup all-acquired ?take-sb'} = \{\} \]

\[ \text{by fact} \]

\[ \text{from dist-prop [rule-format, of 0] t} \]

\[ \text{have dist-t: (all-shared ?take \cup all-unshared ?take \cup all-acquired ?take)} \cap \text{all-shared ?take-sb' \cup all-unshared ?take-sb' \cup all-acquired ?take-sb'} = \{\} \]

\[ \quad \text{apply clarsimp} \]

\[ \text{done} \]

\[ \text{from dist-prop have} \]

\[ \text{dist-prop' } \forall \ i \ p \ is \ O \ R \ D \not\emptyset \ sb. \ i < \text{length ts} \]

\[ \rightarrow ts!i = (p,\text{is,}0,sb,D,O,R) \rightarrow \]

\[ \quad (\text{all-shared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-unshared (takeWhile (Not o is-volatile-Write_{sb}O) sb)} \cup \text{all-acquired (takeWhile (Not o is-volatile-Write_{sb}O) sb)}) \cap \]
apply (clarsimp)
subgoal for i p is O R D \sto sb
apply (drule-tac x=Suc i in spec)
apply clarsimp
done
done
have dist-prop-R-L-A: \forall j p is O R D \sto sb. j < length (t#ts) \implies i \neq j
\rightarrow (t#ts)!j = (p,is,\emptyset,\emptyset,D,O,R) \rightarrow
(all-shared sb \cup all-unshared sb \cup all-acquired sb) \cap
(R \cup L \cup A) = \{\} by fact

from ownership-distinct-tl [OF dist]
have dist\': ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis\': sharing-consis S ts.
then
interpret consis\': sharing-consis S ts .

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent S O t sb t .

from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent S O t (takeWhile (Not \circ is-volatile-Write sb t) sb t).

have aargh: (Not \circ is-volatile-Write sb t) = (\lambda a. \neg is-volatile-Write sb t a)
by (rule ext) auto

show ?case
proof (cases i)
case 0

with t Cons.prems have eqs: p t = p is t = is O t = O R t = R \emptyset t = \emptyset sb t = sb D t = D
by auto

let ?S' = S \oplus_w R \oplus_A L

from dist-prop-R-L-A 0 have
dist-prop-R-L-A:\forall i p is O R D \sto sb. i < length ts
\rightarrow tsli = (p,is,\emptyset,\emptyset,D,O,R) \rightarrow
(all-shared sb \cup all-unshared sb \cup all-acquired sb) \cap
(R \cup L \cup A) = \{\}
apply (clarsimp)
subgoal for i l p is O R D \sto sb
apply (drule-tac x=Suc i l in spec)
apply clarsimp
done
\[
\text{have dist-prop-R-L-A}'' \forall i \ p \is ORD \theta sb. \ i < \text{length ts} \\
\rightarrow \text{ts}!i=(p, \is, \hat{\theta}, \sb, D, O, R) \\
\rightarrow \\
(\text{all-shared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cup \text{all-unshared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cup \\
(\text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb)) \cap \\
(R \cup L \cup A) = \{\} \\
\text{apply (clarsimp)} \\
\text{subgoal for} \ i \ p \is ORD \theta sb \\
\text{apply (cut-tac sb=sb in all-shared-takeWhile-subset [where P=Not } \\
\circ \text{is-volatile-Write}_{sb}]) \\
\text{apply (cut-tac sb=sb in all-unshared-takeWhile-subset [where P=Not } \\
\circ \text{is-volatile-Write}_{sb}]) \\
\text{apply (cut-tac sb=sb in all-acquired-takeWhile-subset [where P=Not } \\
\circ \text{is-volatile-Write}_{sb}]) \\
\text{apply fastforce} \\
\text{done} \\
\text{done}
\]

\[
\text{have sep: } \forall i < \text{length ts}. \\
\text{let } (-, -, -, \sb, -, -, -) = \text{ts} ! i \\
\text{in } \forall a \in \text{all-acquired } sb. \ ?S' \ a = S \ a \\
\text{proof} \quad \{- \\
\quad \text{fix } i \ p_i \is_i \ O_i \ R_i \ D_i \ a_i \sb_i \ a \\
\quad \text{assume i-bound: } i < \text{length ts} \\
\quad \text{assume ts-i: } \text{ts} ! i = (p_i, \is_i, \hat{\theta}_i, \sb_i, D_i, O_i, R_i) \\
\quad \text{assume a-in: } a \in \text{all-acquired } sb_i \\
\quad \text{have } ?S' \ a = S \ a \\
\text{proof} \quad \{- \\
\quad \text{from dist-prop-R-L-A}' [\text{rule-format, OF i-bound ts-i} \ \text{a-in} \\
\quad \text{show } ?\text{thesis} \\
\text{by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)} \\
\text{qed} \\
\} \\
\text{thus } ?\text{thesis by auto} \\
\text{qed} \\
\text{from consis'} \text{sharing-consis-shared-exchange [OF sep]} \\
\text{have sharing-consis'} \text{ sharing-consis } ?S' \ \text{ts.} \\
\text{from share-all-until-volatile-write-share-commute [of } \text{ts} (S \oplus_{W} R \oplus_{A} L) \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb'), \ \text{OF } \text{dist'} \text{sharing-consis'} \text{dist-prop'}\] \\
\text{have share-all-until-volatile-write ts (share } \text{?take-sb'} \ ?S') = \\
\text{share } \text{?take-sb'} (\text{share-all-until-volatile-write ts } ?S') . \\
\text{moreover} \\
\text{from dist-prop-R-L-A}''
\]
have \((\text{share-all-until-volatile-write} \; \text{ts} \; (S \oplus W \; R \ominus A \; L)) = (\text{share-all-until-volatile-write} \; \text{ts} \; S \oplus W \; R \ominus A \; L)\)

apply –
apply \((\text{rule share-all-until-volatile-write-commute} \; \text{OF dist’ consis’, of L A R W, symmetric})\)
apply \((\text{clarsimp blast}+)\)
done
ultimately show ?thesis using takeWhile-empty by (clarsimp simp add: t 0 aargh eqs)

next
case \((\text{Suc} \; k)\)
from Cons.prems Suc obtain k-bound: \(k < \text{length ts}\) and ts-k: \(\text{ts!k} = (p, i, \emptyset, \text{sb, D, O, R})\)
by auto

let \(?S' = (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_{\text{sb}}) \text{sb}_t}) \; S)\)
from freshly-shared-owned \([\text{OF consis-sb}]\)
have fresh-owned: \(\text{dom} \; ?S' - \text{dom} \; S \subseteq O_t\).
from unshared-all-unshared \([\text{OF consis-sb}]\) unshared-acquired-or-owned \([\text{OF consis-sb}]\)
have unshared-acq-owned: \(\text{dom} \; S - \text{dom} \; ?S' \subseteq \text{all-acquired} \; (\text{takeWhile (Not \circ \text{is-volatile-Write}_{\text{sb}}) \text{sb}_t}) \cup O_t\)
by simp

from freshly-shared-owned \([\text{OF consis-sb}]\)
have fresh-owned: \(\text{dom} \; ?S' - \text{dom} \; S \subseteq O_t\).
from unshared-all-unshared \([\text{OF consis-sb}]\) unshared-acquired-or-owned \([\text{OF consis-sb}]\)
have unshared-acq-owned: \(\text{dom} \; S - \text{dom} \; ?S' \subseteq \text{all-acquired} \; (\text{takeWhile (Not \circ \text{is-volatile-Write}_{\text{sb}}) \text{sb}_t}) \cup O_t\)
by simp

have sep:
\(\forall i < \text{length ts}. \; \text{let} \; (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) = \text{ts!i} \in \)
\(\text{all-acquired} \; \text{sb'} \cap \text{dom} \; S - \text{dom} \; ?S' = \{\} \land\)
\(\text{all-unshared} \; \text{sb'} \cap \text{dom} \; ?S' - \text{dom} \; S = \{\}\)
proof –
{
fix i p_i \; i_s; \; \mathcal{O}_i; \; \mathcal{R}_i; \; D_i; \; \text{acq}; \; \emptyset_i; \; \text{sb}_i
assume i-bound: \(i < \text{length ts}\)
assume ts-i: \(\text{ts!i} = (p_i, i_s, \emptyset_i, \text{sb}_i, D_i, \mathcal{O}_i, \mathcal{R}_i)\)
have all-acquired \(\text{sb}_i \cap \text{dom} \; S - \text{dom} \; ?S' = \{\} \land\)
\(\text{all-unshared} \; \text{sb}_i \cap \text{dom} \; ?S' - \text{dom} \; S = \{\}\)
proof –
from ownership-distinct \([\text{of 0 Suc i}]\) ts-i t i-bound
have dist: \((\mathcal{O}_t \cup \text{all-acquired} \; \text{sb}_t) \cap (\mathcal{O}_t \cup \text{all-acquired} \; \text{sb}_t) = \{\}\)
by force

from dist unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-Write\sb) \sb] have all-acquired \sb \cap dom \S − dom ?\S′ = {}
by blast

moreover

from sharing-consis [of Suc i] ts-i i-bound have sharing-consistent \S \O\sb \sb
by force
from unshared-acquired-or-owned [OF this] have all-unshared \sb \subseteq all-acquired \sb \cup \O\sb.
with dist fresh-owned have all-unshared \sb \cap dom ?\S′ − dom \S = {}
by blast

ultimately show ?thesis by simp
  qed
  }
thus ?thesis
by (fastforce simp add: Let-def)
qed
from consis′.sharing-consis-preservation [OF sep] have sharing-consis′: sharing-consis ?\S′ ts.

from dist-prop-R-L-A [rule-format, of 0] Suc t have dist-t-R-L-A: (all-shared \sb \cup all-unshared \sb \cup all-acquired \sb) ∩
(R \cup L \cup A) = {}
apply clarsimp
done
from dist-t-R-L-A have (R \cup L \cup A) \cap (all-shared ?\take \cup all-unshared ?\take \cup all-acquired ?\take) = {}
using all-shared-append [of ?\take ?\drop] all-unshared-append [of ?\take ?\drop] all-acquired-append [of ?\take ?\drop]
by auto

from share-augment-release-commute [OF this] have share ?\take \S \oplus W \R \ominus A \L = share ?\take (\S \oplus W \R \ominus A \L).
moreover

from dist-prop-R-L-A Suc have ∀ j p is \O \R \D \not\sb. j < \text{length} (ts) \rightarrow k \neq j
\rightarrow (ts)!j \=(p, is, @, \sb, \D, \O, \R)
\rightarrow (all-shared \sb \cup all-unshared \sb \cup all-acquired \sb) \cap
(R \cup L \cup A) = {}
apply (clarsimp)
subgoal for j p is \O \R \D \not\sb
apply (drule-tac x=Suc j in spec)
apply clarsimp
done
done

note Cons.hyps [OF dist' sharing-consis' k-bound ts-k dist-prop' this, of W]
ultimately
show ?thesis
  by (clarsimp simp add: t Suc )
qed
qed

lemma share-all-until-volatile-write-Ghost\_sb-commute:
shows \( \bigwedge S j. \) \( \{ \text{ownership-distinct ts; sharing-consis } S \text{ ts; } i < \text{ length ts;} \)
\( \text{ts}[:,i] = (p, is, \theta, \text{Ghost(sb) A L R W#(sb,D,O,R)}) ; \)
\( \forall j \text{ p is } O R D \theta \text{ sb. } j < \text{ length ts } \rightarrow i \neq j \rightarrow \text{ts|j}=(p, is, \theta, D,O,R) \rightarrow \)
\( \text{(all-shared (takeWhile (Not \circ is-volatile-Write\_sb sb) sb) \cup all-unshared (takeWhile (Not \circ is-volatile-Write\_sb sb) sb)) \cap } \)
\( \text{(R \cup L \cup A) = } \{ \} \] 
\( \Rightarrow \)
share-all-until-volatile-write (ts|i :=(p', is', \theta', sb, D', O', R')).) (S \oplus W R \ominus A L) =
share-all-until-volatile-write ts S
proof (induct ts)
case Nil
  thus ?case by simp
next
case (Cons t ts)
  obtain p, is, O, R, D, \theta, \text{sb, where}
  \( t : t=(\text{p,t, \theta,t, D,t, O,t, R,t}) \) 
  by (cases t)
  have dist: ownership-distinct (t\#ts) by fact
  then interpret ownership-distinct t\#ts .
  have consis: sharing-consis S (t\#ts) by fact
  then interpret sharing-consis S t\#ts .
  have dist-prop: \( \forall j \text{ p is } O R D \theta \text{ sb. } j < \text{ length } (t\#ts) \rightarrow i \neq j \rightarrow \)
  \( \text{(t\#ts)|j}=(p, is, \theta, sb, D,O,R) \rightarrow \)
  \( \text{(all-shared (takeWhile (Not \circ is-volatile-Write\_sb sb) sb) \cup all-unshared (takeWhile (Not \circ is-volatile-Write\_sb sb) sb)) \cap } \)
  \( \text{(R \cup L \cup A) = } \{ \} \) by fact
from ownership-distinct-tl [OF dist]
have dist'; ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis'': sharing-consis S ts.
then
interpret consis'': sharing-consis S ts .
from sharing-consis [of 0, simplified, OF t]
have sharing-consistent \( S \mathcal{O}_t \) \( sb_t \).

from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent \( S \mathcal{O}_t \) (takeWhile (Not \( \circ \) is-volatile-Write_{sb}) \( sb_t \)).

let ?\( S' \) = (share (takeWhile (Not \( \circ \) is-volatile-Write_{sb}) \( sb_t \)) \( S \))

from freshly-shared-owned [OF consis-sb]
have fresh-owned: \( \text{dom } ?S' = \text{dom } S \subseteq \mathcal{O}_t \).
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: \( \text{dom } S - \text{dom } ?S' \subseteq \text{all-acquired } \) (takeWhile (Not \( \circ \) is-volatile-Write_{sb}) \( sb_t \)) \( \cup \) \( \mathcal{O}_t \)
  by simp

have sep:
  \( \forall i < \text{length } ts. \) let \((\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = ts!i\) in
  all-acquired \( sb' \cap \text{dom } S - \text{dom } ?S' = \{ \} \wedge \)
  all-unshared \( sb' \cap \text{dom } ?S' - \text{dom } S = \{ \} \)

proof -
  \{  
  fix i \( p_i, is_i, R_i, D_i, O_i, \) \( sb_i \)  
  assume i-bound: \( i < \text{length } ts \)  
  assume ts-i: \( ts!i = (p_i, is_i, \hat{i}, sb_i, D_i, O_i, R_i) \)  
  have all-acquired \( sb_i \cap \text{dom } S - \text{dom } ?S' = \{ \} \wedge \)
  all-unshared \( sb_i \cap \text{dom } ?S' - \text{dom } S = \{ \} \)  
  proof -  
  from ownership-distinct [of 0 Suc i] ts-i t i-bound  
  have dist: \((\mathcal{O}_t \cup \text{all-acquired } sb_i) \cap (\mathcal{O}_t \cup \text{all-acquired } sb_i) = \{ \} \)  
  by force  

from dist unshared-acq-owned all-acquired-takeWhile [of (Not \( \circ \) is-volatile-Write_{sb}) \( sb_i \)]
have all-acquired \( sb_i \cap \text{dom } S - \text{dom } ?S' = \{ \} \)
  by blast

moreover

from sharing-consis [of Suc i] ts-i t i-bound
have sharing-consistent \( S \mathcal{O}_t \) \( sb_i \)
  by force
from unshared-acquired-or-owned [OF this]
have all-unshared \( sb_i \subseteq \text{all-acquired } sb_i \cup \mathcal{O}_t \),
  with dist fresh-owned
have all-unshared \( sb_i \cap \text{dom } ?S' - \text{dom } S = \{ \)  
  by blast

ultimately show \( \)thesis by simp
  qed

345
thus ?thesis
  by (fastforce simp add: Let-def)
qed

from consis'.sharing-consis-preservation [OF sep]
have sharing-consis': sharing-consis (share (takeWhile (Not ◦ is-volatile-Write sb) sb) S) ts.

show ?case
proof (cases i)
  case 0
  with t Cons.prems have eqs: \( p_t = p \), is\_t = is, O\_t = O, R\_t = R, \( \theta_t = \emptyset \), sb\_t = Ghost\_sb, A L R W#sb, D = D
    by auto
  show ?thesis
    by (clarsimp simp add: 0 t eqs)
next
  case (Suc k)
  from Cons.prems Suc obtain k-bound: \( k < \) length ts and ts-k: ts\_k = (p, is, \emptyset, Ghost\_sb, A L R W#sb, D, O, R)
    by auto

  from dist-prop Suc
  have dist-prop': \( \forall \) p is O R D \( \emptyset \) sb. \( j < \) length ts \( \rightarrow k \neq j \rightarrow ts\_j = (p, is, \emptyset, sb, D, O, R) \)
    \( \rightarrow \)
    (all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb) \( \cup \) all-unshared (takeWhile (Not ◦ is-volatile-Write sb) sb) \( \cup \) all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)) \( \cap \)
    (R \( \cup \) L \( \cup \) A) = {\}
    apply (clarsimp)
    subgoal for j p is O R D \( \emptyset \) sb
    apply (drule-tac x=Suc j in spec)
    apply auto
    done
    done

from Cons.hyps [OF dist' sharing-consis' k-bound ts-k dist-prop']
have share-all-until-volatile-write (ts[k := (p', is', \emptyset', sb', D', O', R')])
  (share (takeWhile (Not ◦ is-volatile-Write sb) sb) S \( \oplus_W \) R \( \oplus_A \) L) =
  share-all-until-volatile-write ts
  (share (takeWhile (Not ◦ is-volatile-Write sb) sb) S).

moreover
from dist-prop [rule-format, of 0 p_t = is_t \( \emptyset \) t sb_t D_t O_t R_t] t Suc
  have (R \( \cup \) L \( \cup \) A) \( \cap \) (all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb) \( \cup \) all-unshared (takeWhile (Not ◦ is-volatile-Write sb) sb) \( \cup \) all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)) = {\}

346
apply clarsimp
apply blast
done

done

ultimately
show ?thesis
by (clarsimp simp add: Suc t)
qed

lemma share-all-until-volatile-write-update-sb:
assumes cong: \( \forall S. \text{share} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb') S = \text{share} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) S \)
shows \( \forall S \cdot \forall i. [i < \text{length} ts; ts!i = (p, is, \theta, sb, D, O, R)] \implies \text{share-all-until-volatile-write} ts S = \text{share-all-until-volatile-write} (ts[i := (p', is', \theta', sb', D', O', R')]) S \)
proof (induct ts)
case Nil
thus ?case by simp
next
case (Cons t ts)
obtain p_t is_t O_t R_t \( t \) sb_t where
\( t = (p_t, is_t, \theta_t, sb_t, D_t, O_t, R) \)
by (cases t)

show ?case
proof (cases i)
case 0
with t Cons.prems have eqs: p_t = p is_t = is O_t = O R_t = R \( \theta_t = \theta \) sb_t = sb D_t = D
by auto

show ?thesis
by (clarsimp simp add: 0 t eqs cong)
next
case (Suc k)
from Cons.prems Suc obtain k-bound: k < length ts and ts-k: ts!k = (p, is, \theta, sb, D, O, R)
by auto
from Cons.hyps [OF k-bound ts-k ]
show ?thesis
by (clarsimp simp add: t Suc)
qed

lemma share-all-until-volatile-write-append-Ghost_{sb}:
assumes out-VWrite_{sb}: outstanding-refs is-volatile-Write_{sb} sb \( \neq \{} \)
assumes i-bound: i < length ts
assumes ts-i: \( ts!i = (p, is, \theta, sb, D, O, R) \)

shows share-all-until-volatile-write \( ts \ S = \) share-all-until-volatile-write

\( (ts[i := (p', is', \theta', sb @ [\text{Ghost}\_{ab} A L R W], D', O', R'])] \) \( S \)

proof

from out-VWrite\(_{ab}\)

have \( \bigwedge S. \) share \( \text{takeWhile} (\neg \text{is-volatile-Write}_{ab}) (sb @ [\text{Ghost}_{ab} A L R W]) S = \) share \( \text{takeWhile} (\neg \text{is-volatile-Write}_{ab}) sb S \)

by (simp add: outstanding-vol-write-takeWhile-append)

from share-all-until-volatile-write-update-sb [OF this i-bound ts-i]

show ?thesis

by simp

qed

lemma acquired-append-Prog\(_{ab}\):

\( \bigwedge S. \) acquired pending-write \( \text{takeWhile} (\neg \text{is-volatile-Write}_{ab}) (sb @ [\text{Prog}_{ab} p_1 p_2\ \text{mis}]) S = \)

(already acquired pending-write \( \text{takeWhile} (\neg \text{is-volatile-Write}_{ab}) sb) S \)

by (induct sb) (auto split: memref.splits)

lemma outstanding-refs-non-empty-dropWhile:

outstanding-refs \( P \) \( xs \neq \{\} \implies \) outstanding-refs \( P \) \( \text{dropWhile} (\neg P) \) \( xs \neq \{\} \)

apply (induct \( xs \))
apply simp
apply (simp split: if-split-asm)
done

lemma ex-not: Ex Not

by blast

lemma (in computation) concurrent-step-append:

assumes step: \( (ts,m,S) \Rightarrow (ts',m',S') \)

shows \( (xs@ts,m,S) \Rightarrow (xs@ts',m',S') \)

using step

proof (cases)

case (Program i p is \( \emptyset \) sb \( D O R \) p' is' )

then obtain

i-bound: i < length ts and

ts-i: ts!i = (p, is, \theta, sb, D, O, R) and

qed
prop-step: \( \varnothing \vdash p \rightarrow p \) (p',is') and
ts:\( ts' = ts[i := (p',is@is',\varnothing, record p p', is', sb, D, O, R)] \) and
S': S' = S and
m': m' = m
by auto

from i-bound have i-bound': i + length xs < length (xs@ts)
by auto

from ts-i i-bound have ts-i': (xs@ts)!(i + length xs) = (p, is, \varnothing, sb, D, O, R)
by (auto simp add: nth-append)

from concurrent-step. Program [OF i-bound' ts-i' prog-step, of m S] ts' i-bound
show ?thesis
by (auto simp add: ts' list-update-append S' m')
next
case (Memop i p is sb D O R is' \varnothing sb' D' O' R')
then obtain
i-bound: i < length ts and
ts-i: ts!i = (p, is, \varnothing, sb, D, O, R)
memop-step: (is, \varnothing, sb, m, D, O, R, S) \rightarrow_m (is', \varnothing, sb', m', D', O', R', S') and
ts': ts' = ts[i := (p, is, \varnothing, sb', D', O', R')]
by auto

from i-bound have i-bound': i + length xs < length (xs@ts)
by auto

from ts-i i-bound have ts-i': (xs@ts)!(i + length xs) = (p, is, \varnothing, sb, D, O, R)
by (auto simp add: nth-append)

from concurrent-step. Memop [OF i-bound' ts-i' memop-step] ts' i-bound
show ?thesis
by (auto simp add: ts' list-update-append)
next
case (StoreBuffer i p is sb D O R sb' D' O' R')
then obtain
i-bound: i < length ts and
ts-i: ts!i = (p, is, \varnothing, sb, D, O, R)
sb-step: (m, sb, O, R, S) \rightarrow_{sb} (m', sb', O', R', S') and
ts': ts' = ts[i := (p, is, \varnothing, sb', D', O', R')]
by auto
from i-bound have i-bound': i + length xs < length (xs@ts)
by auto

from ts-i i-bound have ts-i': (xs@ts)!(i + length xs) = (p, is, \varnothing, sb, D, O, R)
by (auto simp add: nth-append)

from concurrent-step. StoreBuffer [OF i-bound' ts-i' sb-step] ts' i-bound
show ?thesis
by (auto simp add: ts' list-update-append)
primrec weak-sharing-consistent:: owns ⇒ 'a memref list ⇒ bool

where
weak-sharing-consistent O [] = True
| weak-sharing-consistent O (r#rs) =
  (case r of
    Write sb volatile - - - A L R W ⇒
      (if volatile then L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ O ∧
       weak-sharing-consistent (O ∪ A − R) rs
      else weak-sharing-consistent O rs)
    | Ghost sb A L R W ⇒
      L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ O ∧ weak-sharing-consistent (O ∪ A − R) rs
    | - ⇒ weak-sharing-consistent O rs)

lemma sharing-consistent-weak-sharing-consistent:
  ⋀ S O. sharing-consistent S O sb ⇒ weak-sharing-consistent O sb
apply (induct sb)
apply (auto split: memref.splits)
done

lemma weak-sharing-consistent-append:
  ⋀ O. weak-sharing-consistent O (xs @ ys) =
  (weak-sharing-consistent O xs ∧ weak-sharing-consistent (acquired True xs O) ys)
apply (induct xs)
apply (auto split: memref.splits)
done

lemma read-only-share-unowned: ⋀ O S.
  [weak-sharing-consistent O sb; a /∈ O ∪ all-acquired sb; a ∈ read-only (share sb S)]
  ⇒ a ∈ read-only S
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case False
with Cons Write sb show ?thesis by auto
next
case True
from Cons.hyps [where S=(S ⊕_W R ⊕_A L) and O=(O ∪ A − R)] Cons.prems
show ?thesis
by (auto simp add: Write sb True in-read-only-restrict-conv in-read-only-augment-conv split: if-split-asm)
qed
next
case \text{Read}_{sb} \textbf{with} \text{Cons} \textbf{show} \ ?\text{thesis} \textbf{by} \text{auto}

next

case \text{Prog}_{sb} \textbf{with} \text{Cons} \textbf{show} \ ?\text{thesis} \textbf{by} \text{auto}

next

case \text{Ghost}_{sb} A L R W

with \text{Cons}\textbf{.hyps} [\textbf{where} S=(S \oplus_{W} R \ominus_{A} L) \textbf{and} O=(O \cup A \setminus R)] \textbf{Cons}\textbf{.prems} \textbf{show} \ ?\text{thesis}

by (\text{auto} \textbf{simp} \textbf{add: in-read-only-restrict-conv in-read-only-augment-conv} \textbf{split: if-split-asm})

\textbf{qed}

\textbf{qed}

\textbf{lemma} \text{share-read-only-mono-in:}

\textbf{assumes} a\text{-}\text{in}: a \in \text{read-only} (\text{share} \text{ sb} \textbf{S}) \textbf{and}

\textbf{assumes} ss: \text{read-only} S \subseteq \text{read-only} S'

\textbf{shows} a \in \text{read-only} (\text{share} \text{ sb} \textbf{S}')

\textbf{using} \textbf{share-read-only-mono} [\textbf{OF} ss] a\text{-}\text{in}

\textbf{by} \text{auto}

\textbf{lemma} \text{read-only-unacquired-share:}

\textbf{\exists} S O. [O \cap \text{read-only} S = \{\} \textbf{; weak-sharing-consistent} O \text{ sb}; a \in \text{read-only} S; a \notin \text{all-acquired} \text{ sb} \]

\Rightarrow a \in \text{read-only} (\text{share} \text{ sb} \textbf{S})

\textbf{proof} (\textbf{induct} \text{ sb})

\textbf{case} Nil \textbf{thus} \ ?\text{case} \textbf{by} \textbf{simp}

\textbf{next}

\textbf{case} (\textbf{Cons} x \text{ sb})

\textbf{show} \ ?\text{case}

\textbf{proof} (\textbf{cases} x)

\textbf{case} (\text{Write}_{\text{sb}} \textbf{volatile} a' \text{ sop} v A L R W)

\textbf{show} \ ?\text{thesis}

\textbf{proof} (\textbf{cases} \text{volatile})

\textbf{case} True

\textbf{note} \text{volatile=this}

\textbf{from} \text{Cons}\textbf{.prems}

\textbf{obtain} a\text{-}\text{ro}: a \in \text{read-only} S \textbf{and} a\text{-A}: a \notin A \textbf{and} a\text{-unacq}: a \notin \text{all-acquired} \text{ sb} \textbf{and}

\text{owns-ro}: O \cap \text{read-only} S = \{\} \textbf{and}

L\text{-A}: L \subseteq A \textbf{and} A\text{-R}: A \cap R = \{\} \textbf{and} R\text{-owns}: R \subseteq O \textbf{and}

\textbf{consis}': weak-sharing-consistent (O \cup A \setminus R) \text{ sb}

\textbf{by} (\text{clarsimp simp add: Write}_{\text{sb}} \text{ True})

\textbf{from} \text{owns-ro A-R owns-ro} \textbf{have} \text{owns-ro}': (O \cup A \setminus R) \cap \text{read-only} (S \oplus_{W} R \ominus_{A} L) = \{\}

\text{351}
by (auto simp add: in-read-only-convs)

  from a-ro a-A owns-ro R-owns L-A have a-ro': a ∈ read-only (S ⊕ W R ⊕ A L)
by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF owns-ro' consis' a-ro' a-unacq]
  show ?thesis
by (clarsimp simp add: Write sb True)
next
  case False
  with Cons show ?thesis
by (clarsimp simp add: Write sb False)
qed
next
  case Read sb with Cons show ?thesis by (clarsimp)
next
  case Prog sb with Cons show ?thesis by (clarsimp)
next
  case (Ghost sb A L R W)
from Cons.prems
obtain a-ro: a ∈ read-only S and a-A: a /∈ A and a-unacq: a /∈ all-acquired sb
  and owns-ro: O ∩ read-only S = {} and
  L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
  consis': weak-sharing-consistent (O ∪ A - R) sb
  by (clarsimp simp add: Ghost sb)

from owns-ro A-R R-owns have owns-ro': (O ∪ A - R) ∩ read-only (S ⊕ W R ⊕ A L)
= {} by (auto simp add: in-read-only-convs)

from a-ro a-A owns-ro R-owns L-A have a-ro': a ∈ read-only (S ⊕ W R ⊕ A L)
by (auto simp add: in-read-only-convs)
from Cons.hyps [OF owns-ro' consis' a-ro' a-unacq]
show ?thesis
by (clarsimp simp add: Ghost sb)
qed
qed

lemma read-only-share-unacquired: ∨ \ O. O ∩ read-only S = {} ⇒
weak-sharing-consistent O sb ⇒
a ∈ read-only (share sb S) ⇒ a /∈ acquired True sb O
proof (induct sb)
  case Nil thus ?case by auto
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a' sop v A L R W)
    show ?thesis

352
proof (cases volatile)
  case False
  with Cons Write<sb> show ?thesis by auto
next
  case True
  note volatile=this
  from Cons.prems
  obtain a-ro: a \in read-only (share sb (S \oplus W R \ominus A L)) and
  owns-ro: O \cap read-only S = \{\} and
  L-A: L \subseteq A and A-R: A \cap R = \{\} and R-owns: R \subseteq O and
  consis': weak-sharing-consistent (O \cup A - R) sb
  by (clarsimp simp add: Write<sb> volatile)

  from owns-ro A-R R-owns have owns-ro': (O \cup A - R) \cap read-only (S \oplus_W R \ominus_A L) = \{
  by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF owns-ro' consis' a-ro]
  show ?thesis
  by (auto simp add: Write<sb> volatile)
qed
next
  case Read<sb> with Cons show ?thesis by auto
next
  case Prog<sb> with Cons show ?thesis by auto
next
  case (Ghost<sb> A L R W)
  from Cons.prems
  obtain a-ro: a \in read-only (share sb (S \oplus_W R \ominus_A L)) and
  owns-ro: O \cap read-only S = \{\} and
  L-A: L \subseteq A and A-R: A \cap R = \{\} and R-owns: R \subseteq O and
  consis': weak-sharing-consistent (O \cup A - R) sb
  by (clarsimp simp add: Ghost<sb>)

  from owns-ro A-R R-owns have owns-ro': (O \cup A - R) \cap read-only (S \oplus_W R \ominus_A L) = \{
  by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF owns-ro' consis' a-ro]
  show ?thesis
  by (auto simp add: Ghost<sb>)
qed
qed

lemma read-only-share-all-acquired-in:
\forall S O. [O \cap read-only S = \{\} ; weak-sharing-consistent O sb ; a \in read-only (share sb S)]

\implies a \in read-only (share sb Map.empty) \lor (a \in read-only S \land a \notin all-acquired sb)

proof (induct sb)
  case Nil thus ?case by simp
next
case (Cons x sb)

show ?case

proof (cases x)
  case (Write sb volatile a' sop v A L R W)

show ?thesis

proof (cases volatile)
  case True

    note volatile=this

  from Cons.prems

    obtain a-in: a ∈ read-only (share sb (S ⊕ W R ⊖ A L)) and owns-ro: O ∩ read-only S = {}

    L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and consis': weak-sharing-consistent (O ∪ A − R) sb

    by (clarsimp simp add: Write sb True)

    from owns-ro A-R owns-ro a-in: owns-ro': (O ∪ A − R) ∩ read-only (S ⊕ W R ⊖ A L) = {}

    by (auto simp add: in-read-only-convs)

    from Cons.hyps [OF owns-ro' consis'a-in]

    have hyp: a ∈ read-only (share sb Map.empty) ∨ a ∈ read-only (S ⊕ W R ⊖ A L) ∧ a ∉ all-acquired sb.

      have a ∈ read-only (share sb (Map.empty ⊕ W R ⊖ A L)) ∨ (a ∈ read-only S ∧ a ∉ A ∧ a ∉ all-acquired sb)

      proof
      { assume a-emp: a ∈ read-only (share sb Map.empty) have read-only Map.empty ⊆ read-only (Map.empty ⊕ W R ⊖ A L)
        by (auto simp add: in-read-only-convs)
      }

      from share-read-only-mono-in [OF a-emp this]

      have a ∈ read-only (share sb (Map.empty ⊕ W R ⊖ A L)).

    } moreover

      { assume a-ro: a ∈ read-only (S ⊕ W R ⊖ A L) and a-unacq: a ∉ all-acquired sb
        have ?thesis

        proof (cases a ∈ read-only S)

          case True

            with a-ro obtain a ∉ A

            by (auto simp add: in-read-only-convs)

          next

            case False

            with a-ro have a-ro-empty: a ∈ read-only (Map.empty ⊕ W R ⊖ A L)

            by (auto simp add: in-read-only-convs split: if-split-asm)

            have read-only (Map.empty ⊕ W R ⊖ A L) ⊆ read-only (S ⊕ W R ⊖ A L)

      }
by (auto simp add: in-read-only-convs)
with owns-ro'
have owns-ro-empty: \((O \cup A - R) \cap \text{read-only} (\text{Map.empty } \oplus_W R \ominus_A L) = \{\}\)
  by blast

from read-only-unacquired-share [OF owns-ro-empty consis' a-ro-empty a-unacq]
have a \in \text{read-only} (\text{share sb} (\text{Map.empty } \oplus_W R \ominus_A L)).
thus ?thesis
  by simp
qed

moreover note hyp
ultimately show ?thesis by blast
qed

then show ?thesis
by (clarsimp simp add: Write_{sb} True)
next
  case False with Cons show ?thesis
by (auto simp add: Write_{sb})
qed
next
  case Read_{sb} with Cons show ?thesis by auto
next
  case Prog_{sb} with Cons show ?thesis by auto
next
  case (\text{Ghost}_{sb} A L R W)
from Cons.prem
obtain a-in: a \in \text{read-only} (\text{share sb} (S \oplus W R \ominus A L)) and
  owns-ro: O \cap \text{read-only} S = \{\} and
  L-A: L \subseteq A and A-R: A \cap R = \{\} and R-owns: R \subseteq O and
  consis': weak-sharing-consistent \((O \cup A - R)\) sb
by (clarsimp simp add: Ghost_{sb})

from owns-ro A-R R-owns have owns-ro': \((O \cup A - R) \cap \text{read-only} (S \oplus_W R \ominus_A L) = \{\}\)
  by (auto simp add: in-read-only-convs)

from Cons.hyps [OF owns-ro' consis'a-in]
have hyp: a \in \text{read-only} (\text{share sb Map.empty}) \lor a \in \text{read-only} (S \oplus_W R \ominus_A L) \land a \notin \text{all-acquired sb}.

have a \in \text{read-only} (\text{share sb} (\text{Map.empty } \oplus_W R \ominus_A L)) \lor (a \in \text{read-only} S \land a \notin A \land a \notin \text{all-acquired sb})
proof -
  { assume a-emp: a \in \text{read-only} (\text{share sb Map.empty})
  have read-only Map.empty \subseteq \text{read-only} (\text{Map.empty } \oplus_W R \ominus_A L)
    by (auto simp add: in-read-only-convs)
from share-read-only-mono-in [OF a-emp this]
have a ∈ read-only (share sb (Map.empty ⊕ W R ⊕ A L)).
  }
moreover
  }
assume a-ro: a ∈ read-only (S ⊕ W R ⊕ A L) and a-unacq: a ∉ all-acquired sb
have ?thesis
  proof (cases a ∈ read-only S)
  case True
  with a-ro obtain a ∉ A
  by (auto simp add: in-read-only-convs)
  with True a-unacq show ?thesis
  by auto
next
  case False
  with a-ro have a-ro-empty: a ∈ read-only (Map.empty ⊕ W R ⊕ A L)
  by (auto simp add: in-read-only-convs split: if-split-asm)

have read-only (Map.empty ⊕ W R ⊕ A L) ⊆ read-only (S ⊕ W R ⊕ A L)
  by (auto simp add: in-read-only-convs)
  with owns-ro'
  have owns-ro-empty: (O ∪ A − R) ∩ read-only (Map.empty ⊕ W R ⊕ A L) = {}
  by blast

from read-only-unacquired-share [OF owns-ro-empty consis’ a-ro-empty a-unacq]
have a ∈ read-only (share sb (Map.empty ⊕ W R ⊕ A L)).
  thus ?thesis
  by simp

qed
}
moreover note hyp
  ultimately show ?thesis by blast
qed
then show ?thesis
  by (clarsimp simp add: Ghost_ab)
qed
qed

lemma weak-sharing-consistent-preserves-distinct:
  ∃ O S. weak-sharing-consistent O sb ⇒ O ∩ read-only S = {} ⇒ acquired True sb O ∩ read-only (share sb S) = {}
proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
show ?case
proof (cases x)
  case (Write sb volatile a sop v A L R W)
  show ?thesis
  proof (cases volatile)
  case True
  note volatile=this
  from Cons.prems obtain
  owns-ro: O ∩ read-only S = {} and L-A: L ⊆ A and A-R: A ∩ R = {} and
  R-owns: R ⊆ O and consis’: weak-sharing-consistent (O ∪ A − R) sb
  by (clarsimp simp add: Write sb True)

  from owns-ro A-R R-owns have owns-ro’: (O ∪ A − R) ∩ read-only (S ⊕ W R ⊕ A L)
  = {}
  by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF consis’ owns-ro’]
  show ?thesis
  by (auto simp add: Write sb True)
next
  case False with Cons Write sb show ?thesis by auto
  qed
next
  case Read sb with Cons show ?thesis by auto
next
  case Prog sb with Cons show ?thesis by auto
next
  case (Ghost sb A L R W)
  from Cons.prems obtain
  owns-ro: O ∩ read-only S = {} and L-A: L ⊆ A and A-R: A ∩ R = {} and
  R-owns: R ⊆ O and consis’: weak-sharing-consistent (O ∪ A − R) sb
  by (clarsimp simp add: Ghost sb)

  from owns-ro A-R R-owns have owns-ro’: (O ∪ A − R) ∩ read-only (S ⊕ W R ⊕ A L)
  = {}
  by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF consis’ owns-ro’]
  show ?thesis
  by (auto simp add: Ghost sb)
  qed
  qed

locale weak-sharing-consis =
fixes ts::('p, p store-buffer, bool, owns, rels) thread-config list
assumes weak-sharing-consis:
\[\forall i \; p \in \mathcal{O} \cap R. D(\theta, sb).\]
\[i < \text{length } ts; ts!i = (p, is, \theta, sb, D, \mathcal{O}, R) \]
\[\Rightarrow\]
weak-sharing-consistent O sb
**sublocale** sharing-consis ⊆ weak-sharing-consis

**proof**
- fix i p is O R D ⊤ sb
- assume i-bound: i < length ts
- assume ts-i: ts ! i = (p, is, ⊤, sb, D, O, R)
  from sharing-consistent-weak-sharing-consistent [OF sharing-consis [OF i-bound ts-i]]
  show weak-sharing-consistent O sb.

qed

**lemma** weak-sharing-consis-tl: weak-sharing-consis (t#ts) ⇒ weak-sharing-consis ts

**apply** (unfold weak-sharing-consis-def)

**apply** force

**done**

**lemma** read-only-share-all-until-volatile-write-unacquired:

∀ S. [ownership-distinct ts; read-only-unowned S ts; weak-sharing-consis ts; ∀ i < length ts. (let (-,-,,-,sb,D,O,R) = ts!i in
  a ∉ all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb));
  a ∈ read-only S]
  ⇒ a ∈ read-only (share-all-until-volatile-write ts S)

**proof** (induct ts)
- case Nil thus ?case by simp
- next
  - case (Cons t ts)
    - obtain p is O R D ⊤ sb where
      t: t = (p,is,⊤,sb,D,O,R)
    - by (cases t)

    have dist: ownership-distinct (t#ts) by fact
    then interpret ownership-distinct t#ts .
    from ownership-distinct-tl [OF dist]
    have dist': ownership-distinct ts.

    have aargh: (Not ◦ is-volatile-Write sb) = (λa. ¬ is-volatile-Write sb a)
    by (rule ext) auto

    have a-ro: a ∈ read-only S by fact
    have ro-unowned: read-only-unowned S (t#ts) by fact
    then interpret read-only-unowned S t#ts .
    have consis: weak-sharing-consis (t#ts) by fact
    then interpret weak-sharing-consis t#ts .

    note consis' = weak-sharing-consis-tl [OF consis]

    let ?take-sb = (takeWhile (Not ◦ is-volatile-Write sb) sb)
    let ?drop-sb = (dropWhile (Not ◦ is-volatile-Write sb) sb)
from weak-sharing-consis [of 0] t
have consis-sb: weak-sharing-consistent \( O \) sb 
by force
with weak-sharing-consistent-append [of \( O \)?take-sb ?drop-sb]
have consis-take: weak-sharing-consistent \( O \)?take-sb 
by auto

have ro-unowned': read-only-unowned (share ?take-sb \( S \)) ts 
proof 
fix j
fix \( p_j \) is_j \( O_j \) \( R_j \) \( D_j \) \( \emptyset_j \) sb_j
assume j-bound: j < length ts
assume jth: ts!j = (\( p_j \) is_j \( \emptyset_j \) sb_j \( D_j \) \( O_j \) \( R_j \))
show \( O_j \) \( \cap \) read-only (share ?take-sb \( S \)) = {}
proof 
\{ 
  fix a 
  assume a-owns: a \( \in \) \( O_j \)
  assume a-ro: a \( \in \) read-only (share ?take-sb \( S \))
  have False
  proof 
  from ownership-distinct [of 0 Suc j] j-bound jth t 
  have dist: (\( O \) \( \cup \) all-acquired sb) \( \cap \) (\( O_j \) \( \cup \) all-acquired sb_j) = {}
  by fastforce 
  from read-only-unowned [of Suc j] j-bound jth 
  have dist-ro: \( O_j \) \( \cap \) read-only \( S \) = {} by force
  show ?thesis
  proof (cases a \( \in \) (\( O \) \( \cup \) all-acquired sb)))
  case True 
  with dist a-owns show False by auto
  next
  case False 
  hence a \( \notin \) (\( O \) \( \cup \) all-acquired ?take-sb)
  using all-acquired-append [of ?take-sb ?drop-sb]
  by auto
  from read-only-share-unowned [OF consis-take this a-ro] 
  have a \( \in \) read-only \( S \).
  with dist-ro a-owns show False by auto
  qed
  qed
\} 
thus ?thesis by auto
qed
qed

from Cons.prems
obtain unacq-ts: \( \forall \) i < length ts. (let (\( \cdot \)\( \cdot \),\( \cdot \),\( \cdot \),\( \cdot \),\( O \)\( \cdot \)) = ts!i in
a \notin \text{all-acquired} \text{ (takeWhile ((Not \circ \text{is-volatile-Write}_{\text{sb}}) \ \text{sb}) \ and \ unacq-sb: \ a \notin \text{all-acquired} \text{ (takeWhile ((Not \circ \text{is-volatile-Write}_{\text{sb}}) \ \text{sb})} \)}

\text{by (force simp add: t aargh)}

\text{from read-only-unowned [of 0] t}
\text{have owns-ro: } \mathcal{O} \cap \text{read-only } \mathcal{S} = \{\}
\text{by force}
\text{from read-only-unacquired-share [OF owns-ro consis-take a-ro unacq-sb]}
\text{have a \in \text{read-only} \ (\text{share (takeWhile ((Not \circ \text{is-volatile-Write}_{\text{sb}}) \ \text{sb}) \ \mathcal{S})}).}
\text{from Cons.hyps [OF dist' ro-unowned' consis' unacq-ts this]}
\text{show ?case}
\text{by (simp add: t)}
\text{qed}

\text{lemma read-only-share-unowned-in:}
[\text{weak-sharing-consistent } \mathcal{O} \ \text{sb; } a \in \text{read-only} \ (\text{share sb } \mathcal{S})]]
\Longrightarrow a \in \text{read-only } \mathcal{S} \cup \mathcal{O} \cup \text{all-acquired sb}
\text{using read-only-share-unowned [of } \mathcal{O} \ \text{sb]}
\text{by auto}

\text{lemma read-only-shared-all-until-volatile-write-subset:}
\bigwedge \mathcal{S}. \ [\text{ownership-distinct ts;}
\quad \text{weak-sharing-consis ts}] \Longrightarrow
\text{read-only (share-all-until-volatile-write ts } \mathcal{S}) \subseteq
\text{read-only } \mathcal{S} \cup (\bigcup ((\lambda (\text{is,-,-,sb,-,O,-}). \mathcal{O} \cup \text{all-acquired} \text{ (takeWhile ((Not } \circ \text{is-volatile-Write}_{\text{sb}}) \ \text{sb})) \setminus \text{ set ts}))}
\text{proof (induct ts)}
\text{case Nil thus ?case by simp}
\text{next}
\text{case (Cons t ts)}
\text{obtain p \text{ is } \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \emptyset \ \text{sb where}}
\text{t: t = (p,is,\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R})}
\text{by (cases t)}
\text{have dist: ownership-distinct (t#ts) by fact}
\text{then interpret ownership-distinct t#ts .}
\text{from ownership-distinct-tl [OF dist]}
\text{have dist': ownership-distinct ts.}

\text{have consis: weak-sharing-consis (t#ts) by fact}
\text{then interpret weak-sharing-consis t#ts .}

\text{have aargh: (Not } \circ \text{is-volatile-Write}_{\text{sb}}) = (\lambda a. \neg \text{is-volatile-Write}_{\text{sb}} a)
\text{by (rule ext) auto}
\text{note consis' = weak-sharing-consis-tl [OF consis]}

360
let ?take-sb = (takeWhile (Not ◦ is-volatile-Write sb) sb)
let ?drop-sb = (dropWhile (Not ◦ is-volatile-Write sb) sb)

from weak-sharing-consis [of 0] t
have consis-sb: weak-sharing-consistent O sb
  by force
with weak-sharing-consistent-append [of O ?take-sb ?drop-sb]
have consis-take: weak-sharing-consistent O ?take-sb
  by auto

{ 
  fix a
  assume a-in: a ∈ read-only
    (share-all-until-volatile-write ts
     (share ?take-sb S)) and
  a-notin-shared: a ∉ read-only S and
  a-notin-rest: a ∉ (⋃((λ(-,-,-,sb,D,O,R). O ∪ all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)) ' set ts))
  have a ∈ O ∪ all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)
  proof –
    from Cons.hyps [OF dist′ consis′, of (share ?take-sb S)] a-in a-notin-rest
    have a ∈ read-only (share ?take-sb S)
      by (auto simp add: aargh)
    from read-only-share-unowned-in [OF consis-take this] a-notin-shared
    show ?thesis by auto
  qed

then show ?case
  by (auto simp add: t aargh)
qed

lemma weak-sharing-consistent-preserves-distinct-share-all-until-volatile-write:
  \forall S. i. [ownership-distinct ts; read-only-unowned S ts;weak-sharing-consis ts; i < length ts; ts!i = (p,is,θ,sb,D,O,R)]
  \implies acquired True (takeWhile (Not ◦ is-volatile-Write sb) sb) O ∩ read-only (share-all-until-volatile-write ts S) = {}

proof (induct ts)
case Nil thus ?case by simp
next
case (Cons t ts)
note read-only-unowned S (t#ts);
then interpret read-only-unowned S t#ts.
note i-bound = i < length (t # ts);
note ith = (t # ts) ! i = (p,is,θ,sb,D,O,R);

have dist: ownership-distinct (t#ts) by fact
then interpret ownership-distinct t#ts.
from ownership-distinct-tl [OF dist]
have dist': ownership-distinct ts.

have consis: weak-sharing-consis (t#ts) by fact
then interpret weak-sharing-consis t#ts.

note consis' = weak-sharing-consis-tl [OF consis]

let ?take-sb = (takeWhile (Not o is-volatile-Write sb) sb)
let ?drop-sb = (dropWhile (Not o is-volatile-Write sb) sb)

have aargh: (Not o is-volatile-Write sb) = (λa. ¬ is-volatile-Write sb a)
by (rule ext) auto
show ?case
proof (cases i)
case 0
from read-only-unowned [of 0] ith 0
have owns-ro: \( \mathcal{O} \cap \text{read-only} \ S = \{\} \)
  by force
from weak-sharing-consis [of 0] ith 0
have weak-sharing-consistent \( \mathcal{O} \) sb
  by force
with weak-sharing-consistent-append [of \( \mathcal{O} \) ?take-sb ?drop-sb]
have consis-take: weak-sharing-consistent \( \mathcal{O} \) ?take-sb
  by auto
from weak-sharing-consistent-preserves-distinct [OF this owns-ro]
have dist-t: acquired True ?take-sb \( \mathcal{O} \cap \text{read-only} \ (\text{share} \ ?take-sb \ S) = \{\} \).
from read-only-shared-all-until-volatile-write-subset [OF dist' consis', of (share ?take-sb S)]
have ro-rest: read-only (share-all-until-volatile-write ts (share ?take-sb S)) ⊆
read-only (\( \bigcup (\lambda(-, -, -, \text{sb}, -, \mathcal{O}, -). \mathcal{O} \cup \text{all-acquired} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write sb}) \text{sb}))' \) set ts))
by auto
{
fix a
assume a-in-sb: a ∈ acquired True ?take-sb \( \mathcal{O} \)
assume a-in-ro: a ∈ read-only (share-all-until-volatile-write ts (share ?take-sb S))
have False
proof –
from Set.in-mono [rule-format, OF ro-rest a-in-ro] dist-t a-in-sb

have a ∈ (\( \bigcup (\lambda(-, -, -, \text{sb}, -, \mathcal{O}, -). \mathcal{O} \cup \text{all-acquired} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write sb}) \text{sb}))' \) set ts))
by auto
then obtain j p j is_j  \( \hat{j} \) sb_j D_j \( \mathcal{O}_j \) R_j
where j-bound: j < length ts and ts-j: ts!j = (p_j is_j  \( \hat{j} \) sb_j D_j \( \mathcal{O}_j \) R_j)
and a-in-j: a ∈ \( \mathcal{O}_j \) ∪ all-acquired (takeWhile (Not \circ is-volatile-Write sb) sb_j)
by (fastforce simp add: in-set-conv-nth)
from ownership-distinct [of 0 Suc j] ith ts-j j-bound 0
have dist: \((\mathcal{O} \cup \text{all-acquired } sb) \cap (\mathcal{O}_j \cup \text{all-acquired } sb_j) = \{\}\)
by fastforce
moreover
from acquired-all-acquired [of True \(?\text{take-sb } \mathcal{O}\) a-in-sb all-acquired-append [of \(?\text{take-sb } \text{drop-sb}\)]
have \(a \in \mathcal{O} \cup \text{all-acquired } sb\)
by auto
with a-in-j all-acquired-append [of (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) j]
(dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)]

have False by fastforce
thus \(?\text{thesis ..}\)
qed

then show \(?\text{thesis}\)
using 0 ith
by (auto simp add: aargh)
next
case (Suc k)
from i-bound Suc have k-bound: \(k < \text{length } ts\)
by auto
from ith Suc have kth: \(ts!k = (p, is, \theta, sb, D, O, R)\)
by auto

obtain \(p_t, is_t, D_t, R_t, \theta_t, sb_t\)
where \(t: t=(p_t, is_t, \theta_t, sb_t, D_t, O_t, R_t)\)
by (cases t)

let \(?\text{take-sb}_t = (\text{takeWhile } \circ \text{is-volatile-Write}_{sb}) sb_t\)
let \(?\text{drop-sb}_t = (\text{dropWhile } \circ \text{is-volatile-Write}_{sb}) sb_t\)

have ro-unowned?: \text{read-only-unowned } (\text{share } ?\text{take-sb}_t S) \text{ ts}
proof
fix j
fix \(p_j, is_j, D_j, R_j, \theta_j, sb_j\)
assume j-bound: \(j < \text{length } ts\)
assume jth: \(ts!j = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j)\)
show \(O_j \cap \text{read-only } (\text{share } ?\text{take-sb}_t S) = \{\}\)
proof
from read-only-unowned [of Suc j] j-bound jth
have dist: \(O_j \cap \text{read-only } S = \{\}\) by force

from weak-sharing-consis [of 0] t
have weak-sharing-consistent \(O_t sb_t\)
by fastforce
with weak-sharing-consistent-append [of \(O_t \text{ ?take-sb}_t \text{ ?drop-sb}_t\)]
have consis-t: weak-sharing-consistent \(O_t ?\text{take-sb}_t\)
by auto
{  
  fix a
  assume a-in-j: a ∈ \(O\)\_j
  assume a-ro: a ∈ \{read-only (share ?take-sb \(S\))\}
  have False
  proof –
  from a-in-j ownership-distinct [of 0 Suc j] j-bound t jth
  have \((O\_t \cup \text{all-acquired} \(sb\_t\)) \cap (O\_j \cup \text{all-acquired} \(sb\_j\)) = \{\}\)
    by fastforce
  with a-in-j all-acquired-append [of ?take-sb \(t\) ?drop-sb \(t\)]
  have a \∉ (\(O\_t \cup \text{all-acquired} \(sb\_t\)))
    by fastforce
  from read-only-share-unowned [OF consis-t this a-ro]
  have a ∈ read-only \(S\).
  with a-in-j dist
  show False by auto
  qed
  }
then
  show ?thesis
  by auto
  qed
  qed

from Cons.hyps [OF dist’ ro-unowned’ consis’ k-bound kth]
  show ?thesis
    by (simp add: t)
  qed
  qed

lemma in-read-only-share-all-until-volatile-write:
  assumes dist: ownership-distinct ts
  assumes consis: sharing-consis \(S\) ts
  assumes ro-unowned: read-only-unowned \(S\) ts
  assumes i-bound: i < length ts
  assumes ts-i: ts!i = (p,is,θ,sb,D,O,R)
  assumes a-unacquired-others: \(\forall j < \text{length ts. } i\neq j \rightarrow\)
    (let \((-,-,sb\_j,-,-,-) = ts\![j]\) in
     a \∉ \text{all-acquired} (takeWhile (Not \(\circ\) is-volatile-Write\(sb\_j\)) \(sb\_j\)))
  assumes a-ro-share: a ∈ read-only (share sb \(S\))
  shows a ∈ read-only (share (dropWhile (Not \(\circ\) is-volatile-Write\(sb\_j\)) sb)
    (share-all-until-volatile-write ts \(S\)))
proof –
  from consis
  interpret sharing-consis \(S\) ts .
  interpret read-only-unowned \(S\) ts by fact

  from sharing-consis [OF i-bound ts-i]
  have consis-sb: sharing-consistent \(S\) \(O\) sb.
from sharing-consistent-weak-sharing-consistent [OF this]
have weak-consis: weak-sharing-consistent $O$ sb.
from read-only-unowned [OF i-bound ts-i]
have owns-ro: $O \cap$ read-only $S = \{}$.
from read-only-share-all-acquired-in [OF owns-ro weak-consis a-ro-share]
have $a \in$ read-only (share sb Map.empty) \lor a \in$ read-only $S \land a \notin$ all-acquired sb.
moreover

let ?take-sb = (takeWhile (Not o is-volatile-Write sb) sb)
let ?drop-sb = (dropWhile (Not o is-volatile-Write sb) sb)

from weak-consis weak-sharing-consistent-append [of $O$ ?take-sb ?drop-sb]
obtain weak-consis': weak-sharing-consistent (acquired True ?take-sb $O$) ?drop-sb and
weak-consis-take: weak-sharing-consistent $O$ ?take-sb
by auto

{
  assume a \in$ read-only (share sb Map.empty)
  with share-append [of ?take-sb ?drop-sb]
  have a-in': a \in$ read-only (share ?drop-sb (share ?take-sb Map.empty))
  by auto

  have owns-empty: $O \cap$ read-only Map.empty = \{}
  by auto

  from weak-sharing-consistent-preserves-distinct [OF weak-consis-take owns-empty]
  have acquired True ?take-sb $O \cap$ read-only (share ?take-sb Map.empty) = \{}.

  from read-only-share-all-acquired-in [OF this weak-consis' a-in']
  have a \in$ read-only (share ?drop-sb Map.empty) \lor a \in$ read-only (share ?take-sb Map.empty) \land a \notin$ all-acquired ?drop-sb.
  moreover
  {
    assume a-ro-drop: a \in$ read-only (share ?drop-sb Map.empty)
    have read-only Map.empty \subseteq$ read-only (share-all-until-volatile-write ts $S$)
    by auto
    from share-read-only-mono-in [OF a-ro-drop this]
    have ?thesis .
  }
  moreover
  {
    assume a-ro-take: a \in$ read-only (share ?take-sb Map.empty)
    assume a-unacq-drop: a \notin$ all-acquired ?drop-sb
    from read-only-share-unowned-in [OF weak-consis-take a-ro-take]
    have $a \in O \cup$ all-acquired ?take-sb by auto
    hence a \in$ O \cup$ all-acquired sb using all-acquired-append [of ?take-sb ?drop-sb]
    by auto
    from share-all-until-volatile-write-thread-local' [OF dist consis i-bound ts-i this] a-ro-share
    have ?thesis by (auto simp add: read-only-def)

  }

}
ultimately have thesis by blast

moreover

{ assume a-ro: a ∈ read-only S
 assume a-unacq: a /∈ all-acquired sb
 with all-acquired-append [of ?take-sb ?drop-sb]
 obtain a /∈ all-acquired ?take-sb and a-notin-drop: a /∈ all-acquired ?drop-sb
 by auto
 with a-unacquired-others i-bound ts-i
 have a-unacq: ∀ j < length ts.
         (let (\_,\_,\_,\_,sb\_j) = ts!j in
           a /∈ all-acquired (takeWhile (Not ◦ is-volatile-Write sb\_j) sb\_j))
 by (auto simp add: Let-def)

 from local.weak-sharing-consis-axioms have weak-sharing-consis ts .
 from read-only-share-all-until-volatile-write-unacquired [OF dist ro-unowned
   \weak-sharing-consis ts: a-unacq a-ro]
 have a-ro-all: a ∈ read-only (share-all-until-volatile-write ts S) .

 from weak-consis weak-sharing-consistent-append [of O ?take-sb ?drop-sb]
 have weak-consis-drop: weak-sharing-consistent (acquired True ?take-sb O) ?drop-sb
 by auto

 from weak-sharing-consistent-preserves-distinct-share-all-until-volatile-write [OF dist ro-unowned \weak-sharing-consis ts: i-bound ts-i]
 have acquired True ?take-sb O ∩
         read-only (share-all-until-volatile-write ts S) = {}.

 from read-only-unacquired-share [OF this weak-consis-drop a-ro-all a-notin-drop]
 have thesis .
}
ultimately show thesis by blast
qed

lemma all-acquired-dropWhile-in: x ∈ all-acquired (dropWhile P sb) ⟹ x ∈ all-acquired sb
using all-acquired-append [of takeWhile P sb dropWhile P sb]
by auto

lemma all-acquired-takeWhile-in: x ∈ all-acquired (takeWhile P sb) ⟹ x ∈ all-acquired sb
using all-acquired-append [of takeWhile P sb dropWhile P sb]
by auto

366
lemmas all-acquired-takeWhile-dropWhile-in = all-acquired-takeWhile-in all-acquired-dropWhile-in

lemma split-in-read-only-reads:
\( \land O. \ a \in \text{read-only-reads} \ O \ xs \implies (\exists t \ v \ ys \ zs. \ xs=\text{ys} @ \text{Read}_{ab} \ False \ a \ t \ v \ # \ zs \land a \notin \text{acquired True} \ ys \ O) \)

proof (induct xs)
  case Nil thus \(?\)case by simp
next
  case (Cons x xs)
  have a-in: a \in \text{read-only-reads} \ O (x#xs) by fact
  show \(?\)case
    proof (cases x)
      case (Write_{ab} volatile a' sop v A L R W)
      show \(?\)thesis
        proof (cases volatile)
          case False
          from a-in have a \in \text{read-only-reads} \ O xs by (auto simp add: Write_{ab} False)
          from Cons.hyps [OF this] obtain t v ys zs where
          xs: xs=\text{ys} @ \text{Read}_{ab} \ False \ a \ t \ v \ # \ zs \land \text{a-notin:} \ a \notin \text{acquired True} \ ys \ O
          by auto
          with xs a-notin obtain x#xs=(x#ys) @ \text{Read}_{ab} \ False \ a \ t \ v \ # \ zs \land \text{a-notin:} \ a \notin \text{acquired True} \ (x#ys) \ O
          by (simp add: Write_{ab} False)
          then show \(?\)thesis
        by blast
      next
      case True
      from a-in have a \in \text{read-only-reads} \ (O \cup A - R) xs by (auto simp add: Write_{ab} True)
      from Cons.hyps [OF this] obtain t v ys zs where
      xs: xs=\text{ys} @ \text{Read}_{ab} \ False \ a \ t \ v \ # \ zs \land \text{a-notin:} \ a \notin \text{acquired True} \ ys \ (O \cup A - R)
      by auto
      with xs a-notin obtain x#xs=(x#ys) @ \text{Read}_{ab} \ False \ a \ t \ v \ # \ zs \land \text{a-notin:} \ a \notin \text{acquired True} \ (x#ys) \ O
      by (simp add: Write_{ab} True)
      then show \(?\)thesis
    by blast
  qed
next
  case (Read_{ab} volatile a't'v')
  show \(?\)thesis
    proof (cases \neg \text{volatile} \land a \notin O \land a'=a)
      case True
      with Read_{ab} show \(?\)thesis
    by fastforce
next

367
case False
  with a-in have a ∈ read-only-reads O xs
by (auto simp add: Read_{sb volatile a - - split: if-split-asm})
  from Cons.hyps [OF this] obtain t v ys zs where
xs: xs=ys@Read_{sb volatile a - -} False a t v # zs and a-notin: a /∈ acquired True ys O
by auto
  with xs a-notin obtain x#xs=(x#ys)@Read_{sb volatile a - -} False a t v # zs a /∈ acquired True (x#ys) O
by (simp add: Read_{sb volatile a - -})
  then show ?thesis
by blast
qed
next
case Prog_{sb volatile a - -}
  with a-in have a ∈ read-only-reads O xs
by (auto)
  from Cons.hyps [OF this] obtain t v ys zs where
xs: xs=ys@Read_{sb volatile a - -} False a t v # zs and a-notin: a /∈ acquired True ys O
by auto
  with xs a-notin obtain x#xs=(x#ys)@Read_{sb volatile a - -} False a t v # zs a /∈ acquired True (x#ys) O
by (simp add: Prog_{sb volatile a - -})
  then show ?thesis
by blast
qed
next
case (Ghost_{sb volatile a - -} A L R W)
  with a-in have a ∈ read-only-reads (O ∪ A - R) xs
by (auto)
  from Cons.hyps [OF this] obtain t v ys zs where
xs: xs=ys@Read_{sb volatile a - -} False a t v # zs and a-notin: a /∈ acquired True ys (O ∪ A - R)
by auto
  with xs a-notin obtain x#xs=(x#ys)@Read_{sb volatile a - -} False a t v # zs a /∈ acquired True (x#ys) O
by (simp add: Ghost_{sb volatile a - -})
  then show ?thesis
by blast
qed

lemma insert-monoD: W ⊆ W′ ⟹ insert a W ⊆ insert a W′
  by blast

primrec unforwarded-non-volatile-reads:: 'a memref list ⇒ addr set ⇒ addr set
where
unforwarded-non-volatile-reads [] W = {}
| unforwarded-non-volatile-reads (x#xs) W =
  (case x of
    Read_{sb volatile a - -} ⇒ (if a /∈ W ∧ ¬ volatile
then insert a (unforwarded-non-volatile-reads xs W)
else (unforwarded-non-volatile-reads xs W))
| Write_{sb} - a - - - - - - ⇒ unforwarded-non-volatile-reads xs (insert a W)
| - ⇒ unforwarded-non-volatile-reads xs W)

**lemma** unforwarded-non-volatile-reads-non-volatile-Read_{sb}:
\[ \forall W. \text{unforwarded-non-volatile-reads } \text{sb } W \subseteq \text{outstanding-refs is-non-volatile-Read}_{sb} \text{ sb} \]
apply (induct sb)
apply (auto split: memref.splits if-split-asm)
done

**lemma** in-unforwarded-non-volatile-reads-non-volatile-Read_{sb}:
a ∈ unforwarded-non-volatile-reads sb W ⇒ a ∈ outstanding-refs is-non-volatile-Read_{sb} sb
using unforwarded-non-volatile-reads-non-volatile-Read_{sb}
by blast

**lemma** unforwarded-non-volatile-reads-antimono:
\[ \forall W, W'. W \subseteq W' \Rightarrow \text{unforwarded-non-volatile-reads } xs W' \subseteq \text{unforwarded-non-volatile-reads } xs W \]
apply (induct xs)
apply (auto split: memref.splits dest: insert-monoD)
done

**lemma** unforwarded-non-volatile-reads-antimono-in:
x ∈ unforwarded-non-volatile-reads xs W' ⇒ W' \subseteq W
⇒ x ∈ unforwarded-non-volatile-reads xs W
using unforwarded-non-volatile-reads-antimono
by blast

**lemma** unforwarded-non-volatile-reads-append:
\[ \forall W. \text{unforwarded-non-volatile-reads } (xs@ys) W = \]
(\text{unforwarded-non-volatile-reads } xs W ∪
unforwarded-non-volatile-reads ys (W ∪ outstanding-refs is-Write_{sb} xs))
apply (induct xs)
apply clasimp
apply (auto split: memref.splits)
done

**lemma** reads-consistent-mem-eq-on-unforwarded-non-volatile-reads:
assumes mem-eq: \( \forall a \in A \cup W. m'a = m a \)
assumes subset: unforwarded-non-volatile-reads sb W ⊆ A
assumes consis-m: reads-consistent pending-write \( O \) m sb
shows reads-consistent pending-write \( O \) m' sb
using mem-eq subset consis-m
proof (induct sb arbitrary: A W m' m pending-write \( O \))
case Nil thus ?case by simp
next
case (Cons r sb)

note mem-eq = (∀ a ∈ A ∪ W. m’ a = m a)

note subset = (unforwarded-non-volatile-reads (r#sb) W ⊆ A)

note consis-m = (reads-consistent pending-write O m (r#sb))

show ?case

proof (cases r)
  case (Write sb volatile a sop v A’ L R W’)
    from subset obtain
      subset’: unforwarded-non-volatile-reads sb (insert a W) ⊆ A
      by (auto simp add: Write sb)
    from mem-eq have mem-eq’:
      ∀ a’ ∈ (A ∪ (insert a W)). (m’(a:=v)) a’ = (m(a:=v)) a’
      by (auto)
    show ?thesis
    proof (cases volatile)
      case True
      from consis-m obtain
        consis’: reads-consistent True (O ∪ A’ − R) (m(a := v)) sb and
        no-volatile-Read sb: outstanding-refs is-volatile-Read sb sb = {}
      by (simp add: Write sb True)
      from Cons.hyps [OF mem-eq’ subset’ consis’]
      have reads-consistent True (O ∪ A’ − R) (m’(a := v)) sb.
      with no-volatile-Read sb
      show ?thesis
    by (simp add: Write sb True)
    next
      case False
      from consis-m obtain consis’: reads-consistent pending-write O (m(a := v)) sb
      by (simp add: Write sb False)
      from Cons.hyps [OF mem-eq’ subset’ consis’]
      have reads-consistent pending-write O (m’(a := v)) sb.
      then
      show ?thesis
    by (simp add: Write sb False)
  qed

next
  case (Read sb volatile a t v)
  from mem-eq have mem-eq’:
    ∀ a’ ∈ A ∪ W. m’ a’ = m a’
    by (auto)
  show ?thesis
  proof (cases volatile)
    case True
    note volatile=this
    from consis-m obtain
      consis’: reads-consistent pending-write O m sb
by (simp add: Read sb True)

  show ?thesis
  proof (cases a ∈ W)
  case False
  from subset obtain
    subset': unforwarded-non-volatile-reads sb W ⊆ A
    using False
    by (auto simp add: Read sb True split: if-split-asm)
  from Cons.hyps [OF mem-eq' subset' consis']
  show ?thesis
    by (simp add: Read sb True)
  next
  case True
  from subset have
    subset': unforwarded-non-volatile-reads sb W ⊆ insert a A
    using True
    apply (auto simp add: Read sb volatile split: if-split-asm)
    done
  from mem-eq True have mem-eq': ∀ a' ∈ (insert a A) ∪ W. m' a' = m a'
    by auto
  from Cons.hyps [OF mem-eq' subset' consis']
  show ?thesis
    by (simp add: Read sb volatile)
  qed
  next
  case False
  note non-vol = this
  from consis-m obtain
    consis': reads-consistent pending-write O m sb and
    v: (pending-write ∨ a ∈ O) → v = m a
  by (simp add: Read sb False)
  show ?thesis
    proof (cases a ∈ W)
    case True
    from mem-eq subset Read sb True non-vol have m' a = m a
    by (auto simp add: False)
    from subset obtain
      subset': unforwarded-non-volatile-reads sb W ⊆ insert a A
      using False
      by (auto simp add: Read sb non-vol split: if-split-asm)
    from mem-eq True have mem-eq': ∀ a' ∈ (insert a A) ∪ W. m' a' = m a'
      by auto
    with Cons.hyps [OF mem-eq' subset' consis'] v
    show ?thesis
      by (simp add: Read sb non-vol)
    next
  case False

371
from mem-eq subset Read sb False non-vol have meq: m' a = m a
   by (clarsimp )
from subset obtain
  subset': unforwarded-non-volatile-reads sb W ⊆ A
    using non-vol False
    by (auto simp add: Read sb non-vol split: if-split-asm)
from mem-eq non-vol have mem-eq': ∀ a' ∈ A ∪ W. m' a' = m a'
   by auto
with Cons.hyps [OF mem-eq' subset' consis'] v meq
show ?thesis
   by (simp add: Read sb non-vol False)
qed
qed
next
case Prog sb with Cons show ?thesis by auto
next
case Ghost sb with Cons show ?thesis by auto
qed
qed

lemma reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop:
  assumes mem-eq: ∀ a ∈ A ∪ W. m' a = m a
  assumes subset: unforwarded-non-volatile-reads (dropWhile (Not ◦ is-volatile-Write sb) sb) W ⊆ A
  assumes subset-acq: acquired-reads True (takeWhile (Not ◦ is-volatile-Write sb) sb) O ⊆ A
  assumes consis-m: reads-consistent False O m sb
  shows reads-consistent False O m' sb
using mem-eq subset subset-acq consis-m
proof (induct sb arbitrary: A W m' m O)
case Nil thus ?case by simp
next
case (Cons r sb)
  note mem-eq = ∀ a ∈ A ∪ W. m' a = m a
  note subset = unforwarded-non-volatile-reads (dropWhile (Not ◦ is-volatile-Write sb) (r#sb)) W ⊆ A
  note subset-acq = acquired-reads True (takeWhile (Not ◦ is-volatile-Write sb) (r#sb)) O ⊆ A
  note consis-m = reads-consistent False O m (r#sb)
show ?case
proof (cases r)
  case (Write sb volatile a sop v A' L R W')
  show ?thesis
  proof (cases volatile)
    case True
    from mem-eq
    have mem-eq':
      ∀ a' ∈ (A ∪ (insert a W)). (m'(a:=v)) a' = (m(a:=v)) a'
by (auto)

from consis-m obtain
consis': reads-consistent True \((O \cup A' - R) (m(a := v)) sb\) and
no-volatile-Read\(_{sb}\): outstanding-refs is-volatile-Read\(_{sb}\) sb = {}
by (simp add: Write\(_{sb}\) True)

from subset obtain unforwarded-non-volatile-reads sb (insert a W) \(\subseteq A\)
by (clarsimp simp add: Write\(_{sb}\) True)

from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads [OF mem-eq' this consis']
have reads-consistent True \((O \cup A' - R) (m'(a := v)) sb\).
with no-volatile-Read\(_{sb}\)
show ?thesis
by (simp add: Write\(_{sb}\) True)
next
case False
from mem-eq
have mem-eq':
\[\forall a' \in (A \cup W). (m'(a := v)) a' = (m(a := v)) a'\]
by (auto)
from subset obtain
subset': unforwarded-non-volatile-reads (dropWhile (Not ◦ is-volatile-Write\(_{sb}\)) sb) W \(\subseteq A\)
by (auto simp add: Write\(_{sb}\) False)
from subset-acq have
subset-acq': acquired-reads True (takeWhile (Not ◦ is-volatile-Write\(_{sb}\)) sb) \(O \subseteq A\)
by (auto simp add: Write\(_{sb}\) False)

from consis-m obtain consis': reads-consistent False \(O (m(a := v)) sb\)
by (simp add: Write\(_{sb}\) False)
from Cons.hyps [OF mem-eq' subset' subset-acq' consis']
have reads-consistent False \(O (m'(a := v)) sb\).
then
show ?thesis
by (simp add: Write\(_{sb}\) False)
qed

next
case (Read\(_{sb}\) volatile a t v)
from mem-eq
have mem-eq':
\[\forall a' \in A \cup W. m' a' = m a'\]
by (auto)
from subset obtain
subset': unforwarded-non-volatile-reads (dropWhile (Not ◦ is-volatile-Write\(_{sb}\)) sb) W \(\subseteq A\)
by (clarsimp simp add: Read\(_{sb}\))
from subset-acq obtain
a-A: \(\neg\) volatile \(\longrightarrow a \in O \longrightarrow a \in A\) and
subset-acq': acquired-reads True (takeWhile (Not o is-volatile-Write sb) sb) \( O \subseteq A \)
by (auto simp add: Read sb split: if-split-asm)

show \(?thesis

proof (cases volatile)
  case True
  note volatile=this
  from consis-m obtain consis': reads-consistent False \( O \ m \ sb \)
  by (simp add: Read sb True)
    from Cons.hyps [OF mem-eq' subset' subset-acq' consis']
  show ?thesis
  next
  case False
  note non-vol = this
  from consis-m obtain consis': reads-consistent False \( O \ m \ sb \) and
  v: a \( \in \) \( O \rightarrow v=m a \)
  by (simp add: Read sb False)
    from mem-eq a-A v have v': a \( \in \) \( O \rightarrow v=m' a \)
  by (auto simp add: non-vol)
    from Cons.hyps [OF mem-eq' subset' subset-acq' consis'] v'
  show ?thesis
  qed
next
  case Prog sb with Cons show \(?thesis by auto
next
  case Ghost sb with Cons show \(?thesis by auto
qed
qed

lemma read-only-read-witness:
\[ \forall S O. \]
\[
[\text{non-volatile-owned-or-read-only True } S O \ sb; \]
\[ a \in \text{read-only-reads } O \ sb] \]
\[ \implies \exists x s y v. \ \text{sb}=xs@ \text{Read sb False } a \ t \ v \ # \ y s \ \wedge \ a \in \text{read-only } (\text{share } x s \ S) \ \wedge \ a \notin \text{read-only-reads } O \ xs
\]

proof (induct sb)
  case Nil thus \(?case by simp
next
  case (Cons x sb)
  show \(?case

374
proof (cases x)
  case (Write\textsubscript{sb} volatile a\textquotesingle sop v A L R W)
  show \textasciitilde thesis
  proof (cases volatile)
    case True

    from Cons.prems obtain
      a-ro: a \in \text{read-only-reads} (O \cup A - R) sb and
      nvo\textquotesingle: \text{non-volatile-owned-or-read-only} True (S \oplus W R \ominus A L) (O \cup A - R) sb
      by (clarsimp simp add: Write\textsubscript{sb} True)

      from Cons.hyps [OF nvo\textquotesingle a-ro]
      obtain xs ys t v where
        sb = xs @ Read\textsubscript{sb} False a t v \# ys \land a \in \text{read-only} (share xs (S \oplus W R \ominus A L)) \land
        a \notin \text{read-only-reads} (O \cup A - R) xs
      by blast

      thus \textasciitilde thesis
      apply --
      apply (rule-tac x=(x#xs) in exI)
      apply (rule-tac x=ys in exI)
      apply (rule-tac x=t in exI)
      apply (rule-tac x=v in exI)
      apply (clarsimp simp add: Write\textsubscript{sb} True)
    done
    next
    case False
    from Cons.prems obtain
      a-ro: a \in \text{read-only-reads} O sb and
      nvo\textquotesingle: \text{non-volatile-owned-or-read-only} True S O sb
      by (clarsimp simp add: Write\textsubscript{sb} False)

      from Cons.hyps [OF nvo\textquotesingle a-ro]
      obtain xs ys t v where
        sb = xs @ Read\textsubscript{sb} False a t v \# ys \land a \in \text{read-only} (share xs S) \land a \notin \text{read-only-reads} O xs
      by blast

      thus \textasciitilde thesis
      apply --
      apply (rule-tac x=(x#xs) in exI)
      apply (rule-tac x=ys in exI)
      apply (rule-tac x=t in exI)
      apply (rule-tac x=v in exI)
      apply (clarsimp simp add: Write\textsubscript{sb} False)
    done
    qed
  next
  case (Read\textsubscript{sb} volatile a\textquotesingle t v)
  show \textasciitilde thesis
proof (cases a'\models a \land a \notin O \land \neg \text{volatile})
  case True
  with Cons.prems have a \in \text{read-only } S
  by (simp add: Read_{sb})

  with True show \?thesis
    apply –
    apply (rule-tac x=[] in exI)
    apply (rule-tac x=sb in exI)
    apply (rule-tac x=t in exI)
    apply (rule-tac x=v in exI)
    apply (clarsimp simp add: Read_{sb})
  done
next
  case False
  with Cons.prems obtain
    a-ro: a \in \text{read-only-reads } O \text{ sb and }
    nvo': non-volatile-owned-or-read-only True S O sb
  by (auto simp add: Read_{sb} split: if-split-asm)
  from Cons.hyps [OF nvo' a-ro]
  obtain xs ys t' v' where
    sb = xs @ Read_{sb} False a t' v' \# ys \land a \in \text{read-only (share xs } S) \land a \notin \text{read-only-reads } O \text{ xs}
  by blast

  with False show \?thesis
    apply –
    apply (rule-tac x=(x#xs) in exI)
    apply (rule-tac x=ys in exI)
    apply (rule-tac x=t' in exI)
    apply (rule-tac x=v' in exI)
    apply (clarsimp simp add: Read_{sb} )
  done
qed

next
  case Prog_{sb}
  from Cons.prems obtain
    a-ro: a \in \text{read-only-reads } O \text{ sb and }
    nvo': non-volatile-owned-or-read-only True S O sb
  by (clarsimp simp add: Prog_{sb})

  from Cons.hyps [OF nvo' a-ro]
  obtain xs ys t v where
    sb = xs @ Read_{sb} False a t v \# ys \land a \in \text{read-only (share xs } S) \land a \notin \text{read-only-reads } O \text{ xs}
  by blast

thus \?thesis
  apply –
  apply (rule-tac x=(x#xs) in exI)

376
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Prog)
done

next
case (Ghost sb A L R W)
from Cons.prems obtain
  a-ro: a ∈ read-only-reads (O ∪ A − R) sb and
  nvo’: non-volatile-owned-or-read-only True (S ⊕ W R ⊕ A L) (O ∪ A − R) sb
  by (clarsimp simp add: Ghost)

  from Cons.hyps [OF nvo’ a-ro]
  obtain xs ys t v where
    sb = xs @ Read sb False a t v # ys ∧ a ∈ read-only (share xs (S ⊕ W R ⊕ A L)) ∧ a /∈
    read-only-reads (O ∪ A − R) xs
    by blast

  thus ?thesis
  apply –
  apply (rule-tac x=(x#xs) in exI)
  apply (rule-tac x=ys in exI)
  apply (rule-tac x=t in exI)
  apply (rule-tac x=v in exI)
  apply (clarsimp simp add: Ghost)
done
qed

lemma read-only-read-acquired-witness: ∀S O.
  [non-volatile-owned-or-read-only True S O sb; sharing-consistent S O sb;
   a /∈ read-only S; a /∈ O; a ∈ read-only-reads O sb] 
  ⊨
    ∃xs ys t v. sb=xs@ Read sb False a t v # ys ∧ a ∈ all-acquired xs ∧ a ∈ read-only (share
    xs S) ∧
    a /∈ read-only-reads O xs
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a’ sop v A L R W)
show ?thesis
proof (cases volatile)
  case True
  note volatile=\ this
  from Cons.prems obtain
    nvo’: non-volatile-owned-or-read-only True (S ⊕ W R ⊕ A L) (O ∪ A − R) sb and
a-nro: \( a \notin \text{read-only } S \) and

a-unowned: \( a \notin O \) and

a-ro\': \( a \in \text{read-only-reads } (O \cup A \setminus R) \) sb and

A-shared-owns: \( A \subseteq \text{dom } S \cup O \) and L-A: \( L \subseteq A \) and A-R: \( A \cap R = \{\} \) and

R-owns: \( R \subseteq O \) and

consis\': sharing-consistent \( (S \oplus_{W} R \ominus_{A} L) (O \cup A \setminus R) \) sb
by (clarsimp simp add: Writea_{sb} True)

from R-owns a-unowned
have a-R: \( a \notin R \)
by auto

show \( ?\)thesis
proof (cases \( a \in A \))
case True
from read-only-read-witness \([OF \text{nvo' a-ro}'\]
obtain \( xs \ ys \ t \ v' \) where
\( sb: sb = xs @ \text{Read}_{a_{sb}} \) False \( a \ t \ v' \# \ ys \) and
\( ro: a \in \text{read-only } (\text{share } xs (S \oplus_{W} R \ominus_{A} L)) \) and
\( a-ro-xs: a \notin \text{read-only-reads } (O \cup A \setminus R) \) xs
by blast

with True show \( ?\)thesis
apply –
apply (rule-tac \( x=x#xs \) in \( exI \))
apply (rule-tac \( x=ys \) in \( exI \))
apply (rule-tac \( x=t \) in \( exI \))
apply (rule-tac \( x=v' \) in \( exI \))
apply (clarsimp simp add: Writea_{sb} volatile)
done

next
case False
with a-unowned R-owns a-nro L-A A-R
obtain a-nro\': \( a \notin \text{read-only } (S \oplus_{W} R \ominus_{A} L) \) and a-unowned\': \( a \notin O \cup A \setminus R \)
by (force simp add: in-read-only-convs)

from Cons.hyps \([OF \text{nvo' consis' a-nro'} a-unowned' a-ro']\)
obtain \( xs \ ys \ t \ v' \) where \( sb = xs @ \text{Read}_{a_{sb}} \) False \( a \ t \ v' \# \ ys \) and
\( a \in \text{all-acquired } xs \wedge a \in \text{read-only } (\text{share } xs (S \oplus_{W} R \ominus_{A} L)) \) and
\( a \notin \text{read-only-reads } (O \cup A \setminus R) \) xs
by blast

then show \( ?\)thesis
apply –
apply (rule-tac \( x=x#xs \) in \( exI \))
apply (rule-tac \( x=ys \) in \( exI \))
apply (rule-tac \( x=t \) in \( exI \))
apply (rule-tac \( x=v' \) in \( exI \))
apply (clarsimp simp add: Writea_{sb} volatile)
done
qed
next
case False
from Cons.prems obtain
consis': sharing-consistent \( S \ O \) \( sb \) and
a-nro': \( a \notin \) read-only \( S \) and
a-unowned: \( a \notin \ O \) and
a-ro': \( a \in \) read-only-reads \( O \) \( sb \) and
\( a' \in \ O \) and
nvo': non-volatile-owned-or-read-only True \( S \ O \) \( sb \)
by (clarsimp simp add: Write\( sb \) False)

from Cons.hyps [OF nvo' consis' a-nro' a-unowned a-ro']
obtain xs ys t v' where
\( sb = xs @ \text{Read}_{sb} \text{False} \ a \ t \ v' \# \ ys \ \wedge \)
\( a \in \) all-acquired \( xs \) \( \wedge \ a \in \) read-only (share \( xs \) \( S \)) \( \wedge \ a \notin \) read-only-reads \( O \) \( xs \)
by blast

then show ?thesis
apply –
apply (rule-tac x=x#xs in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v' in exI)
apply (clarsimp simp add: Read\( sb \) False)
done
qed
next
case (Read\( sb \) volatile \( a' \ t \ v)\)
from Cons.prems
obtain
consis': sharing-consistent \( S \ O \) \( sb \) and
a-nro': \( a \notin \) read-only \( S \) and
a-unowned: \( a \notin \ O \) and
a-ro': \( a \in \) read-only-reads \( O \) \( sb \) and
nvo': non-volatile-owned-or-read-only True \( S \ O \) \( sb \)
by (auto simp add: Read\( sb \) split: if-split-asm)

from Cons.hyps [OF nvo' consis' a-nro' a-unowned a-ro']
obtain xs ys t v' where
\( sb = xs @ \text{Read}_{sb} \text{False} \ a \ t \ v' \# \ ys \ \wedge \)
\( a \in \) all-acquired \( xs \) \( \wedge \ a \in \) read-only (share \( xs \) \( S \)) \( \wedge \ a \notin \) read-only-reads \( O \) \( xs \)
by blast

with Cons.prems show ?thesis
apply –
apply (rule-tac x=x#xs in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v' in exI)
apply (clarsimp simp add: Read\( sb \))

379
proof

apply (clarsimp simp add: Prog sb)

then show ?thesis

proof (cases a ∈ A)

next

from Cons.prems obtain

obtain consis': sharing-consistent $S \circ \sigma$ sb and

next

case Prog sb

from Cons.prems obtain

obtain consis': sharing-consistent $S \circ \sigma$ sb and

a-nro': $a \notin$ read-only $S$ and

a-unowned: $a \notin \sigma$ and

a-ro': $a \in$ read-only-reads $\sigma$ sb and

nvo': non-volatile-owned-or-read-only True $S \circ \sigma$ sb

by (auto simp add: Prog sb)

from Cons.hyps [OF nvo' consis' a-nro' a-unowned a-ro']

obtain xs ys t v where

sb = xs @ Read sb False a t v # ys and

a ∈ all-acquired xs ∧ a ∈ read-only (share xs $S$) ∧ a ∈ read-only-reads $\sigma$ xs

by blast

then show ?thesis

apply -

apply (rule-tac x=x#xs in exI)

apply (rule-tac x=ys in exI)

apply (rule-tac x=t in exI)

apply (rule-tac x=v in exI)

apply (clarsimp simp add: Prog sb)

done

next

case (Ghost sb A L R W)

from Cons.prems obtain

obtain consis': sharing-consistent $(S \oplus W R \ominus_A L) (\sigma_\cup \setminus \setminus A \setminus R) sb$ and

a-nro: $a \notin$ read-only $S$ and

a-unowned: $a \notin \sigma$ and

a-ro': $a \in$ read-only-reads $(\sigma_\cup \setminus \setminus A \setminus R) sb$ and

A-shared-owns: $A \subseteq \text{dom } S \cup \sigma$ and L-A: $L \subseteq A$ and A-R: $A \cap R = \{\}$ and

R-owns: $R \subseteq \sigma$ and

nvo': non-volatile-owned-or-read-only True $(S \oplus W R \ominus_A L) (\sigma_\cup \setminus \setminus A \setminus R) sb$

by (clarsimp simp add: Ghost sb)

from R-owns a-unowned

have a-R: $a \notin R$

by auto

show ?thesis

proof (cases a ∈ A)

case True

from read-only-read-witness [OF nvo' a-ro']

obtain xs ys t v' where

sb: sb = xs @ Read sb False a t v' # ys and

ro: a ∈ read-only (share xs $(S \oplus W R \ominus_A L)$) and

a-ro-xs: $a \notin$ read-only-reads $(\sigma_\cup \setminus \setminus A \setminus R) xs$

by blast

380
with True show \( \text{thesis} \)

apply —
apply (rule-tac \( x=x\#xs \) in exI)
apply (rule-tac \( x=ys \) in exI)
apply (rule-tac \( x=t \) in exI)
apply (rule-tac \( x=v' \) in exI)
apply (clarsimp simp add: Ghost\_sb)
done

next
case False
with a-unowned R-owns a-nro L-A A-R
obtain a-nro': a \( \notin \) read-only \( (S \oplus W R \ominus A L) \) and a-unowned': a \( \notin \) \( O \cup A - R \)
by (force simp add: in-read-only-convs)

from Cons\_hyps [OF nvo' consis' a-nro' a-unowned' a-ro']
obtain xs ys t v' where sb = xs @ Read\_sb False a t v' # ys \&
a \in all-acquired xs \& a \in read-only (share xs (S \oplus W R \ominus A L)) \& 
a \notin read-only-reads \( (O \cup A - R) \) xs
by blast

then show \( \text{thesis} \)
apply —
apply (rule-tac \( x=x\#xs \) in exI)
apply (rule-tac \( x=ys \) in exI)
apply (rule-tac \( x=t \) in exI)
apply (rule-tac \( x=v' \) in exI)
apply (clarsimp simp add: Ghost\_sb)
done
qed

lemma unforwarded-not-written: \( \forall W. a \in \text{unforwarded-non-volatile-reads sb W} \implies a \notin W \)
proof (induct sb)
case Nil thus \( ?\text{case} \) by simp
next
case (Cons \( x \) sb)
show \( ?\text{case} \)
proof (cases x)
case (Write\_sb volatile a' sop v A L R W')
from Cons\_prems
have a \in unforwarded-non-volatile-reads sb (insert a' W)
by (clarsimp simp add: Write\_sb)
from Cons\_hyps [OF this]
have a \notin insert a' W.
then show \( \text{thesis} \)
by blast
next
case (Read\sb, volatile a' t v)
  with Cons.hyps [of W] Cons.prems show ?thesis
  by (auto split: if-split-asm)
next
case Prog\sb
  with Cons.hyps [of W] Cons.prems show ?thesis
  by (auto split: if-split-asm)
next
case Ghost\sb
  with Cons.hyps [of W] Cons.prems show ?thesis
  by (auto split: if-split-asm)
qed
qed

lemma unforwarded-witness:\A X.
  [a \in unforwarded-non-volatile-reads sb X]
  \implies
  \exists xs ys t v. sb=xs@ Read\sb False a t v # ys \land a \notin outstanding-refs is-Write\sb xs
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
  show ?case
  proof (cases x)
  case (Write\sb, volatile a'sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True

      from Cons.prems obtain
      a-unforw: a \in unforwarded-non-volatile-reads sb (insert a' X)
      by (clarsimp simp add: Write\sb True)

      from unforwarded-not-written [OF a-unforw]
      have a'-a: a'\noteq a
      by auto

      from Cons.hyps [OF a-unforw]
      obtain xs ys t v where
      sb = xs @ Read\sb False a t v # ys \land
      a \notin outstanding-refs is-Write\sb xs
      by blast

      thus ?thesis
      using a'-a
      apply -
      apply (rule-tac x=(x#xs) in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Write \_sb True)
done

next
case False
from Cons.prems obtain
a-unforw: a ∈ unforwarded-non-volatile-reads sb (insert a’ X)
by (clarsimp simp add: Write \_sb False)

from unforwarded-not-written [OF a-unforw]
have a’-a: a’\neq a
by auto

from Cons.hyps [OF a-unforw]
obtain xs ys t v where
sb = xs @ Read \_sb False a t v # ys ∧
a \notin outstanding-refs is-Write \_sb xs
by blast

thus ?thesis
using a’-a
apply –
apply (rule-tac x=(x#xs) in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Read \_sb False)
done
qed

next
case (Read \_sb volatile a’ t v)
show ?thesis
proof (cases a’=a ∧ a \notin X ∧ ¬ volatile)
case True

with True show ?thesis
apply –
apply (rule-tac x=[] in exI)
apply (rule-tac x=sb in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Read \_sb)
done

next
case False
note not-ror = this
with Cons.prems obtain a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Read \_sb split: if-split-asm)
from Cons.hyps [OF a-unforw]
obtain xs ys t v where
sb = xs @ Readsb False a t v # ys ∧
a /∈ outstanding-refs is-Write
xs by blast

thus ?thesis
apply –
apply (rule-tac x=(x#xs) in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Read)
done
qed

next
case Progsb
from Cons.prems obtain a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Progsb)

from Cons.hyps [OF a-unforw]
obtain xs ys t v where
sb = xs @ Readsb False a t v # ys ∧
a /∈ outstanding-refs is-Write xs
by blast

thus ?thesis
apply –
apply (rule-tac x=(x#xs) in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Progsb)
done

next
case (Ghostsb A L R W)
from Cons.prems obtain a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Ghostsb)

from Cons.hyps [OF a-unforw]
obtain xs ys t v where
sb = xs @ Readsb False a t v # ys ∧
a /∈ outstanding-refs is-Write xs
by blast

thus ?thesis
apply –
apply (rule-tac x=(x#xs) in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v in exI)
apply (clarsimp simp add: Ghost sb)
done
qed
qed

lemma read-only-read-acquired-unforwarded-witness: \( \forall S O X. \)
\[ (\nonvolatile-owned-or-read-only \text{ True } S O sb; \ \text{sharing-consistent } S O sb; \]
a \notin \text{ read-only } S; \ a \notin O; \ a \in \text{ read-only-reads } O sb;
a \in \text{ unforwarded-non-volatile-reads } sb X \]
\[ \implies \exists xs ys t v. \ sb=xs@\text{Read}_sb \ False a t v \ # \ ys \ \wedge \ a \in \text{ all-acquired } xs \wedge \]
a \notin \text{ outstanding-refs is-Write}_sb xs

proof (induct sb)
  case (Nil)
  thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a' sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True
      note volatile=this
      fix Cons.prems obtain
      nvo': \nonvolatile-owned-or-read-only \text{ True } (S \oplus W R \ominus A L) (O \cup A - R) \text{ sb and }
a-nro: a \notin \text{ read-only } S \text{ and }
a-unowned: a \notin O \text{ and }
a-ro': a \in \text{ read-only-reads } (O \cup A - R) \text{ sb and }
A-shared-owns: A \subseteq \text{ dom } S \cup O \text{ and } L-A: L \subseteq A \text{ and } A-R: A \cap R = \{\} \text{ and }
R-owns: R \subseteq O \text{ and }
consis': \text{ sharing-consistent } (S \oplus W R \ominus A L) (O \cup A - R) \text{ sb and }
a-unforw: a \in \text{ unforwarded-non-volatile-reads } sb \text{ (insert } a' X) \text{ by (clarsimp simp add: Write}_sb \text{ True})

    from unforwarded-not-written [OF a-unforw]
    have a-notin: a \notin \text{ insert } a' X.
    from R-owns a-unowned
    have a-R: a \notin R
    by auto
    show ?thesis
    proof (cases a \in A)
      case True

from unforwarded-witness [OF a-unforw]
obtain xs ys t v' where
  sb: sb = xs @ \text{Read}_sb False a t v' \ # \ ys \text{ and }
a-xs: a \notin \text{ outstanding-refs is-Write}_sb xs

385
by blast

with True a-notin show ?thesis
  apply –
  apply (rule-tac x=x#xs in exI)
  apply (rule-tac x=ys in exI)
  apply (rule-tac x=t in exI)
  apply (rule-tac x=v’ in exI)
  apply (clarsimp simp add: Write sb volatile)
  done
next
case False
with a-unowned R-owns a-nro L-A A-R
obtain a-nro’: a $\notin$ read-only ($S \oplus W R \ominus A L$) and a-unowned’: a $\notin$ $O \cup A - R$
by (force simp add: in-read-only-convs)

from Cons.hyps [OF nvo’ consis’ a-nro’ a-unowned’ a-ro’ a-unforw]
obtain xs ys t v’ where sb = xs @ Read sb False a t v’ # ys ∧
a $\in$ all-acquired xs ∧
a $\notin$ outstanding-refs is-Write sb xs
by blast

with a-notin show ?thesis
  apply –
  apply (rule-tac x=x#xs in exI)
  apply (rule-tac x=ys in exI)
  apply (rule-tac x=t in exI)
  apply (rule-tac x=v’ in exI)
  apply (clarsimp simp add: Write sb volatile)
  done
  qed
next
case False
from Cons.prems obtain
  consis’: sharing-consistent $S O sb$ and
  a-nro’: a $\notin$ read-only $S$ and
  a-unowned: a $\notin$ $O$ and
  a-ro’: a $\in$ read-only-reads $O sb$ and
  a’ $\in$ $O$ and
  nvo’: non-volatile-owned-or-read-only True $S O sb$ and
  a-unforw’: a $\in$ unforwarded-non-volatile-reads sb (insert a’ X)
by (auto simp add: Write sb False split: if-split-asm)

  from unforwarded-not-written [OF a-unforw’]
  have a-notin: a $\notin$ insert a’ X.

  from Cons.hyps [OF nvo’ consis’ a-nro’ a-unowned a-ro’ a-unforw’]
  obtain xs ys t v’ where
  sb = xs @ Read sb False a t v’ # ys ∧
a $\in$ all-acquired xs ∧ a $\notin$ outstanding-refs is-Write sb xs
by blast

  with a-notin show ?thesis
apply –
apply (rule-tac x=x#xs in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v’ in exI)
apply (clarsimp simp add: Write sb False)
done
qed
next
case (Read sb volatile a’ t v)
from Cons.prems
obtain
  consis’: sharing-consistent S O sb and
  a-nro’: a /∈ read-only S and
  a-unowned: a /∈ O and
  a-ro’: a ∈ read-only-reads O sb and
  nvo’: non-volatile-owned-or-read-only True S O sb and
  a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Read sb split: if-split-asm)

from Cons.hyps [OF nvo’ consis’ a-nro’ a-unowned a-ro’ a-unforw]
obtain xs ys t v’ where
  sb = xs @ Read sb False a t v’ # ys ∧
  a ∈ all-acquired xs ∧ a /∈ outstanding-refs is-Write sb xs
by blast

with Cons.prems show ?thesis
apply –
apply (rule-tac x=x#xs in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v’ in exI)
apply (clarsimp simp add: Read sb)
done
next
case Prog sb
from Cons.prems
obtain
  consis’: sharing-consistent S O sb and
  a-nro’: a /∈ read-only S and
  a-unowned: a /∈ O and
  a-ro’: a ∈ read-only-reads O sb and
  nvo’: non-volatile-owned-or-read-only True S O sb and
  a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Prog sb)

from Cons.hyps [OF nvo’ consis’ a-nro’ a-unowned a-ro’ a-unforw]
obtain \( xs \) \( ys \) \( t \) \( v \) where
\[
\begin{align*}
sb &= xs @ \text{Read}_{sb} \text{ False a t v} \neq ys \\
a &\in \text{all-acquired} \hspace{1mm} xs \land a \notin \text{outstanding-refs is-Write}_{sb} \hspace{1mm} xs
\end{align*}
\]
by blast

then show \(?\)thesis
apply –
apply \((\text{rule-tac} \hspace{1mm} x=x\#xs \hspace{1mm} \text{in exI})\)
apply \((\text{rule-tac} \hspace{1mm} x=ys \hspace{1mm} \text{in exI})\)
apply \((\text{rule-tac} \hspace{1mm} x=t \hspace{1mm} \text{in exI})\)
apply \((\text{rule-tac} \hspace{1mm} x=v \hspace{1mm} \text{in exI})\)
apply \((\text{clarsimp simp add: Prog}_{sb})\)
done

next
case \((\text{Ghost}_{sb} \hspace{1mm} A \hspace{1mm} L \hspace{1mm} R \hspace{1mm} W)\)
from \(\text{Cons.prems obtain}\)
\begin{itemize}
\item nvo': \(\text{non-volatile-owned-or-read-only} \hspace{1mm} \text{True} \hspace{1mm} (S \oplus W \hspace{1mm} R \hspace{1mm} \ominus A, L) \hspace{1mm} (O \cup A - R) \hspace{1mm} sb \text{ and}\)
\item a-nro: \(a \notin \text{read-only} \hspace{1mm} S \hspace{1mm} \text{and}\)
\item a-unowned: \(a \notin O \hspace{1mm} \text{and}\)
\item a-ro': \(a \in \text{read-only-reads} \hspace{1mm} (O \cup A - R) \hspace{1mm} sb \text{ and}\)
\item A-shared-owns: \(A \subseteq \text{dom} \hspace{1mm} S \cup O \hspace{1mm} \text{and} \hspace{1mm} L-A \hspace{1mm} \subseteq \hspace{1mm} A \hspace{1mm} \text{and} \hspace{1mm} A-R \hspace{1mm} : \hspace{1mm} A \cap R = \{\} \hspace{1mm} \text{and}\)
\item R-owns: \(R \subseteq O \hspace{1mm} \text{and}\)
\item consis': \(\text{sharing-consistent} \hspace{1mm} (S \oplus W \hspace{1mm} R \hspace{1mm} \ominus A, L) \hspace{1mm} (O \cup A - R) \hspace{1mm} sb \hspace{1mm} \text{and}\)
\item a-unforw: \(a \in \text{unforwarded-non-volatile-reads} \hspace{1mm} sb \hspace{1mm} (X)\)
\end{itemize}
by \((\text{clarsimp simp add: Ghost}_{sb})\)

from \(\text{unforwarded-not-written} \hspace{1mm} \text{[OF a-unforw]}\)
have a-notin: \(a \notin X.\)
from \(\text{R-owns a-unowned}\)
have a-R: \(a \notin R \text{ and}\)
by auto
show \(?\)thesis
proof \((\text{cases} \hspace{1mm} a \in A)\)
\begin{itemize}
\item case True
\end{itemize}

from \(\text{unforwarded-witness} \hspace{1mm} \text{[OF a-unforw]}\)
obtain \( xs \) \( ys \) \( t \) \( v' \) where
\[
\begin{align*}
sb &= sb = xs @ \text{Read}_{ab} \text{ False a t v'} \neq ys \hspace{1mm} \text{and}
\end{align*}
\]
a-xs: \(a \notin \text{outstanding-refs is-Write}_{sb} \hspace{1mm} xs \hspace{1mm} \text{by blast}\)

with \(\text{True a-notin}\)
show \(?\)thesis
apply –
apply \((\text{rule-tac} \hspace{1mm} x=x\#xs \hspace{1mm} \text{in exI})\)
apply \((\text{rule-tac} \hspace{1mm} x=ys \hspace{1mm} \text{in exI})\)
apply \((\text{rule-tac} \hspace{1mm} x=t \hspace{1mm} \text{in exI})\)
apply \((\text{rule-tac} \hspace{1mm} x=v' \hspace{1mm} \text{in exI})\)
apply \((\text{clarsimp simp add: Ghost}_{sb})\)
done
next

388
case False
with a-unowned R-owns a-nro L-A A-R
obtain a-nro': a /∈ read-only (S ⊕ W R ⊕ A L) and a-unowned': a /∈ O ∪ A − R
by (force simp add: in-read-only-convs)
from Cons.hyps [OF nvo' consis' a-nro' a-unowned' a-ro' a-unforw]
obtain xs ys t v' where sb = xs ⊕ Read_a'b False a t v' # ys ∧
a ∈ all-acquired xs ∧
a /∈ outstanding-refs is-Write_a'b xs
by blast
with a-notin show ?thesis
apply ~
apply (rule-tac x=x#xs in exI)
apply (rule-tac x=ys in exI)
apply (rule-tac x=t in exI)
apply (rule-tac x=v' in exI)
apply (clarsimp simp add: Ghost_a'b)
done
qed
qed

lemma takeWhile-prefix: ∃ ys. takeWhile P xs @ ys = xs
apply (induct xs)
apply auto
done

lemma unforwarded-empty-extend:
∀ W. x ∈ unforwarded-non-volatile-reads sb {} ⇒ x /∈ W ⇒ x ∈ unforwarded-non-volatile-reads sb W
apply (induct sb)
apply clarsimp
subgoal for a sb W
apply (case-tac a)
apply clarsimp
apply (frule unforwarded-not-written)
apply (drule-tac W={} in unforwarded-non-volatile-reads-antimono-in)
apply blast
apply (auto split: if-split-asm)
done
done

lemma notin-unforwarded-empty:
∀ W. a /∈ unforwarded-non-volatile-reads sb W ⇒ a /∈ W ⇒ a /∈ unforwarded-non-volatile-reads sb {}
using unforwarded-empty-extend
by blast
lemma
assumes ro: \(a \in \text{read-only } S \rightarrow \) \(a \in \text{read-only } S'\)
assumes a-in: \(a \in \text{read-only } (S \oplus W R)\)
shows a \(\in \text{read-only } (S' \oplus W R)\)
using ro a-in
by (auto simp add: in-read-only-convs)

lemma
assumes ro: \(a \in \text{read-only } S \rightarrow \) \(a \in \text{read-only } S'\)
assumes a-in: \(a \in \text{read-only } (S \ominus A L)\)
shows a \(\in \text{read-only } (S' \ominus A L)\)
using ro a-in
by (auto simp add: in-read-only-convs)

lemma non-volatile-owned-or-read-only-read-only-reads-eq:
\[\forall S S' O \text{ pending-write}.\]
\[\forall a \in \text{read-only-reads } O \text{ sb}. \ a \in \text{read-only } S \rightarrow \ a \in \text{read-only } S'\]
\[\Rightarrow \text{non-volatile-owned-or-read-only pending-write } S' O \text{ sb}\]
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True
note volatile=this
from Cons.prems obtain
nvo': non-volatile-owned-or-read-only-read-only True (S \oplus W R \ominus A L) (O \cup A \ominus R) sb and
ro': \(\forall a \in \text{read-only-reads } (O \cup A \ominus R) \text{ sb}. \ a \in \text{read-only } S \rightarrow \ a \in \text{read-only } S'\)
by (clarsimp simp add: Write sb volatile)

from ro'
have ro'': \(\forall a \in \text{read-only-reads } (O \cup A \ominus R) \text{ sb}.\)
\(a \in \text{read-only } (S \oplus W R \ominus A L) \rightarrow \ a \in \text{read-only } (S' \oplus W R \ominus A L)\)
by (auto simp add: in-read-only-convs)
from Cons.prems obtain nvo'' ro''
show ?thesis
by (clarsimp simp add: Write sb volatile)
next
case False
with Cons.prems [of pending-write S O S'] Cons.prems show ?thesis
by (auto simp add: Write sb)
qed
next
case (Read sb volatile a t v)
show ?thesis
proof (cases volatile)
  case True
    with Cons.hyps [of pending-write $S \circ S'$] Cons.prems show ?thesis
    by (auto simp add: Read$_{ab}$)
  next
  case False
    note non-vol = this
    show ?thesis
    proof (cases $a \in O$)
      case True
      with Cons.hyps [of pending-write $S \circ S'$] Cons.prems show ?thesis
      by (auto simp add: Read$_{ab}$ non-vol)
    next
    case False
    from Cons.prems Cons.hyps [of pending-write $S \circ S'$] show ?thesis
    by (clarsimp simp add: Read$_{ab}$ non-vol False)
  qed
qed

next
  case Prog$_{ab}$
  with Cons.hyps [of pending-write $S \circ S'$] Cons.prems show ?thesis
  by (auto)
next
  case (Ghost$_{ab}$ $A R W$)
  from Cons.hyps [of pending-write ($S \oplus W \ominus A \ominus L \oplus W \ominus A L$) $O \cup A \ominus R S'$] Cons.prems show ?thesis
  by (auto simp add: Ghost$_{ab}$ in-read-only-convs)
qed

lemma non-volatile-owned-or-read-only-read-only-reads-eq':
$
\land_{S, S'} O.
[\text{non-volatile-owned-or-read-only False } S \circ S \circ b;\]
\forall a \in \text{read-only-reads} \ (\text{acquired True} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{ab}) \ sb) \ O)
\ (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{ab}) \ sb). \ a \in \text{read-only} \ S \longrightarrow a \in \text{read-only} \ S'\]
\implies \text{non-volatile-owned-or-read-only False } S' \circ O \circ b$

proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write$_{ab}$ volatile $a$ sop $v \ A R W$)
    show ?thesis
    proof (cases volatile)
      case True
  
391
\begin{verbatim}

note volatile=\textit{this}

from Cons.prems obtain

nvo': non-volatile-owned-or-read-only True \((S \oplus W \ominus A L) (O \cup A \setminus R) sb\) and

ro': \(\forall a \in\text{read-only-reads} (O \cup A \setminus R) sb. a \in\text{read-only} S \rightarrow a \in\text{read-only} S'\)

by (clarsimp simp add: \texttt{Write$_{\text{av}}$ volatile})

from ro'

have ro'': \(\forall a \in\text{read-only-reads} (O \cup A \setminus R) sb.

\(a \in\text{read-only} (S \oplus W R \ominus A L) \rightarrow a \in\text{read-only} (S' \oplus W R \ominus A L)\)

by (auto simp add: in-read-only-convs)

from non-volatile-owned-or-read-only-read-only-reads-eq [OF nvo' ro'']

show \(?\text{thesis}\)

by (clarsimp simp add: \texttt{Write$_{\text{av}}$ volatile})

next

case False

with Cons.hyps [of \(S O S'\)] Cons.prems show \(?\text{thesis}\)

by (auto simp add: \texttt{Read$_{\text{av}}$})

qed

next

case (\texttt{Read$_{\text{av}}$ volatile a t v})

show \(?\text{thesis}\)

proof (cases volatile)

\textit{case} True

with Cons.hyps [of \(S O S'\)] Cons.prems show \(?\text{thesis}\)

by (auto simp add: \texttt{Read$_{\text{av}}$ non-vol})

next

case False

from Cons.prems Cons.hyps [of \(S O S'\)] show \(?\text{thesis}\)

by (clarsimp simp add: \texttt{\textit{Read$_{\text{av}}$ non-vol False}})

qed

next

case \texttt{Prog$_{\text{av}}$}

with Cons.hyps [of \(S O S'\)] Cons.prems show \(?\text{thesis}\)

by (auto)

next

case \texttt{Ghost$_{\text{av}}$ A L R W}

from Cons.hyps [of \((S \oplus W R \ominus A L) O \cup A \setminus R S' \oplus W R \ominus A L\)] Cons.prems

show \(?\text{thesis}\)

by (auto simp add: \texttt{\textit{Ghost$_{\text{av}}$ in-read-only-convs}})

qed

qed

\end{verbatim}
lemma no-write-to-read-only-memory-read-only-reads-eq:
\[ \forall S S'. \]
[ no-write-to-read-only-memory \( S \) \( \text{sb} \); \( \forall a \in \text{outstanding-refs is-Write}_{sb} \) \( a \in \text{read-only} \) \( S' \) \( \rightarrow \) \( a \in \text{read-only} \) \( S \) ]
\[ \Longrightarrow \text{no-write-to-read-only-memory} \ S' \ \text{sb} \]
proof (induct \( \text{sb} \))
case Nil thus ?case by simp
next
  case (Cons \( x \) \( \text{sb} \))
  show ?case
  proof (cases \( x \))
    case (Write\(_{sb}\) \( \text{volatile} \) \( a \) \( \text{sop} v A L W \))
    show ?thesis
    proof (cases \( \text{volatile} \))
      case True
      note \( \text{volatile}=\text{this} \)
      from Cons.prems obtain
      nvo': \( \text{no-write-to-read-only-memory} \ (S \oplus W R \ominus A L) \) \( \text{sb} \) and
      ro': \( \forall a\in\text{outstanding-refs is-Write}_{sb} \) \( a \in \text{read-only} \) \( S' \) \( \rightarrow \) \( a \in \text{read-only} \) \( S \) and
      not-ro: \( a \notin \text{read-only} \) \( S' \)
      by (auto simp add: Write\(_{sb}\) \( \text{volatile} \))
      from ro'
      have ro'': \( \forall a\in\text{outstanding-refs is-Write}_{sb} \) \( a \in \text{read-only} \) \( S' \) \( \rightarrow \) \( a \in \text{read-only} \) \( S \) and
      not-ro: \( a \notin \text{read-only} \) \( S' \)
      by (auto simp add: in-read-only-convs)
      from Cons.hyps [OF nvo' ro'' not-ro]
      show ?thesis
      by (clarsimp simp add: Write\(_{sb}\) \( \text{volatile} \))
    next
      case False
      with Cons.hyps [of \( S \) \( S' \)] Cons.prems show ?thesis
    by (auto simp add: Read\(_{sb}\))
    qed
  next
    case (Read\(_{sb}\) \( \text{volatile} \) \( a t v \))
    with Cons.hyps [of \( S \) \( S' \)] Cons.prems show ?thesis
    by (auto simp add: Read\(_{sb}\))
  next
    case Progs\(_{sb}\)
    with Cons.hyps [of \( S \) \( S' \)] Cons.prems show ?thesis
    by (auto)
  next
    case (Ghost\(_{sb}\) \( A L R W \))
    from Cons.hyps [of \( S \oplus W R \ominus A L \) \( S' \oplus W R \ominus A L \)] Cons.prems
    show ?thesis
    by (auto simp add: Ghost\(_{sb}\) in-read-only-convs)
lemma reads-consistent-drop:
reads-consistent False $O$ m sb  
$\implies$ reads-consistent True
  (acquired True (takeWhile (Not is-volatile-Write sb) sb) $O$)
  (flush (takeWhile (Not is-volatile-Write sb) sb) m)
  (dropWhile (Not is-volatile-Write sb) sb)
using reads-consistent-append [of False $O$ m (takeWhile (Not is-volatile-Write sb) sb)]
apply (cases outstanding-refs is-volatile-Write sb = {})
apply (clarsimp simp add: outstanding-vol-write-take-drop-appends)
apply (fastforce simp add: outstanding-refs-conv)
apply (frule dropWhile-ConsD)
apply (clarsimp split: memref.splits)
done

lemma outstanding-refs-non-volatile-Readsb-all-acquired-dropWhile':
$\forall m \ S \ O$ pending-write.
  [reads-consistent pending-write $O$ m sb; non-volatile-owned-or-read-only pending-write $S$ $O$ sb;]
a $\in$ outstanding-refs is-non-volatile-Readsb (dropWhile (Not is-volatile-Write sb) sb)]
$\implies$ a $\in$ $O$ $\lor$ a $\in$ all-acquired sb $\lor$
  a $\in$ read-only-reads (acquired True (takeWhile (Not is-volatile-Write sb) sb) $O$)
  (dropWhile (Not is-volatile-Write sb) sb)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write sb volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case True
  note volatile=this
from Cons.prems obtain
non-vo: non-volatile-owned-or-read-only True ($S \oplus_R R \ominus_A L$)
  ($O \cup A \ominus R$) sb and
out-vol: outstanding-refs is-volatile-Readsb sb = {} and
out: a $\in$ outstanding-refs is-non-volatile-Readsb sb
by (clarsimp simp add: Write sb True)
  show ?thesis
proof (cases \( a \in \mathcal{O} \))

\begin{enumerate}
  \item \textbf{case} True
    \begin{enumerate}
      \item \textbf{show} \(?\text{thesis}\)
        \begin{enumerate}
          \item \textbf{by} (clarsimp simp add: Write\(_{sb}\) True volatile)
        \end{enumerate}
      \end{enumerate}

  \item \textbf{case} False
    \begin{enumerate}
      \item \textbf{from} \text{outstanding-non-volatile-Read}_{sb}-acquired-or-read-only-reads [OF \text{non-vo out}]
      \item \textbf{have} a-in: \( a \in \text{acquired-reads True sb } (\mathcal{O} \cup A - R) \lor \)
        \( a \in \text{read-only-reads } (\mathcal{O} \cup A - R) \) sb
        \begin{enumerate}
          \item \textbf{by} auto
        \end{enumerate}
      \item \textbf{with} \text{acquired-reads-all-acquired [of True sb } (\mathcal{O} \cup A - R)]
      \item \textbf{show} \(?\text{thesis}\)
        \begin{enumerate}
          \item \textbf{by} (auto simp add: Write\(_{sb}\) volatile)
        \end{enumerate}
    \end{enumerate}

  \item \textbf{next}
  \begin{enumerate}
    \item \textbf{case} False
      \begin{enumerate}
        \item \textbf{with} Cons \textbf{show} \(?\text{thesis}\)
          \begin{enumerate}
            \item \textbf{by} (auto simp add: Write\(_{sb}\) False)
          \end{enumerate}
      \end{enumerate}
  \end{enumerate}

  \item \textbf{next}
  \begin{enumerate}
    \item \textbf{case} Read_{sb}
      \begin{enumerate}
        \item \textbf{with} Cons \textbf{show} \(?\text{thesis}\)
          \begin{enumerate}
            \item \textbf{apply} (clarsimp simp del: o-apply simp add: Read\(_{sb}\) acquired-takeWhile-non-volatile-Write\(_{sb}\) split: if-split-asm)
            \item \textbf{apply} auto
            \item \textbf{done}
          \end{enumerate}
      \end{enumerate}
  \end{enumerate}

  \item \textbf{next}
  \begin{enumerate}
    \item \textbf{case} Prog_{sb}
      \begin{enumerate}
        \item \textbf{with} Cons \textbf{show} \(?\text{thesis}\)
          \begin{enumerate}
            \item \textbf{by} (auto simp add: Read_{sb})
          \end{enumerate}
      \end{enumerate}
  \end{enumerate}

  \item \textbf{next}
  \begin{enumerate}
    \item \textbf{case} (\text{Ghost}_{sb} A L R W)
      \begin{enumerate}
        \item \textbf{with} Cons.hyps \[ \text{of pending-write } \mathcal{O} \cup A - R \text{ m } S \oplus_W R \ominus_A L \]
        \item \textbf{read-only-reads-antimono [of } \mathcal{O} \mathcal{O} \cup A - R \]
        \item \textbf{Cons.prems} \textbf{show} \(?\text{thesis}\)
          \begin{enumerate}
            \item \textbf{by} (auto simp add: \text{Ghost}_{sb})
          \end{enumerate}
      \end{enumerate}
  \end{enumerate}

  \item \textbf{qed}
\end{enumerate}

end

theory ReduceStoreBufferSimulation
imports ReduceStoreBuffer
begin

locale initial_{sb} = simple-ownership-distinct + read-only-unowned + unowned-shared +

395
proof
  i j p
fix
  using
  empty-sb
assumes
  empty-is
assumes
  constrains
  ts
  : (\{ p.p store-buffer, bool.owns, rels \}) thread-config list
assumes empty-sb: \[ i < length ts; ts! = (p, is, xs, sb, D, O, R) ]\Rightarrow sb = []\]
assumes empty-is: \[ i < length ts; ts! = (p, is, xs, sb, D, O, R) ]\Rightarrow is = []\]
assumes empty-rels: \[ i < length ts; ts! = (p, is, xs, sb, D, O, R) ]\Rightarrow R = Map.empty

sublocale initial_i_b \subseteq outstanding-non-volatile-refs-owned-or-read-only
proof
  fix i is O R D \theta sb p
  assume i-bound: \[ i < length ts \]
  assume ts-i: ts! = (p, is, \theta, sb, D, O, R)
  show non-volatile-owned-or-read-only False S O sb
  using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial_i_b \subseteq outstanding-volatile-writes-unowned-by-others
proof
  fix i j p i_s O i_R D \theta sb p_j i_s O_j i_R D_j \theta_j sb_j
  assume i-bound: \[ i < length ts \]
  j-bound: \[ j < length ts \]
  neq-i-j: \[ i \neq j \]
  ts-i: \[ ts! = (p_i, is, \theta_i, sb_i, D_i, O_i, R_i) \]
  ts-j: \[ ts! = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) \]
  show (O_i \cup all-acquired sb_i) \cap outstanding-refs is-volatile-Write_i_b sb_i = {}
qed

sublocale initial_i_b \subseteq read-only-reads-unowned
proof
  fix i j p i_s O i_R D \theta sb p_j i_s O_j i_R D_j \theta_j sb_j
  assume i-bound: \[ i < length ts \]
  j-bound: \[ j < length ts \]
  neq-i-j: \[ i \neq j \]
  ts-i: \[ ts! = (p_i, is, \theta_i, sb_i, D_i, O_i, R_i) \]
  ts-j: \[ ts! = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) \]
  show (O_i \cup all-acquired sb_i) \cap
       read-only-reads (acquired True
                        (takeWhile (Not o is-volatile-Write_i_b sb_i) O_i)
                        (dropWhile (Not o is-volatile-Write_i_b sb_i) sb_i) ) = {}
qed

sublocale initial_i_b \subseteq ownership-distinct
proof
  fix i j p i_s O i_R D \theta sb p_j i_s O_j i_R D_j \theta_j sb_j
  assume i-bound: \[ i < length ts \]
  j-bound: \[ j < length ts \]
  neq-i-j: \[ i \neq j \]
  ts-i: \[ ts! = (p_i, is, \theta_i, sb_i, D_i, O_i, R_i) \]
  ts-j: \[ ts! = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) \]
  show (O_i \cup all-acquired sb_i) \cap (O_j \cup all-acquired sb_j) = {}
qed

sublocale initial_i_b \subseteq valid-ownership ..

sublocale initial_i_b \subseteq outstanding-non-volatile-writes-unshared
proof


fix \( i \) is \( O R D \theta sb p \)
assume i-bound: \( i < \) length ts
assume ts-i: \( ts_i = (p,is,\theta sb,D,O,R) \)
show non-volatile-writes-unshared \( S sb \)
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial_{sb} \subseteq sharing-consis
proof
fix i is \( O R D \theta sb p \)
assume i-bound: \( i < \) length ts
assume ts-i: \( ts_i = (p,is,\theta sb,D,O,R) \)
show sharing-consistent \( S O sb \)
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial_{sb} \subseteq no-outstanding-write-to-read-only-memory
proof
fix i is \( O R D \theta sb p \)
assume i-bound: \( i < \) length ts
assume ts-i: \( ts_i = (p,is,\theta sb,D,O,R) \)
show no-write-to-read-only-memory \( S sb \)
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial_{sb} \subseteq valid-sharing ..
sublocale initial_{sb} \subseteq valid-ownership-and-sharing ..

sublocale initial_{sb} \subseteq load-tmps-distinct
proof
fix i is \( O R D \theta sb p \)
assume i-bound: \( i < \) length ts
assume ts-i: \( ts_i = (p,is,\theta sb,D,O,R) \)
show distinct-load-tmps is
using empty-is [OF i-bound ts-i] by auto
qed

sublocale initial_{sb} \subseteq read-tmps-distinct
proof
fix i is \( O R D \theta sb p \)
assume i-bound: \( i < \) length ts
assume ts-i: \( ts_i = (p,is,\theta sb,D,O,R) \)
show distinct-read-tmps sb
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial_{sb} \subseteq load-tmps-read-tmps-distinct
proof
fix i is \( O R D \theta sb p \)
assume i-bound: \( i < \) length ts
assume ts-i: \( ts_i = (p,is,\theta sb,D,O,R) \)
show load-tmps is \( \cap \) read-tmps sb = \{\}
using empty-sb [OF i-bound ts-i] empty-is [OF i-bound ts-i] by auto
qed

sublocale initial_{sb} \subseteq load-tmps-read-tmps-distinct ..

sublocale initial_{sb} \subseteq valid-write-sops
proof
fix i is \( O R D \theta sb p \)
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
show ∀sop ∈ write-sops sb. valid-sop sop
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ valid-store-sops
proof
  fix i is O R D θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
  show ∀sop ∈ store-sops is. valid-sop sop
  using empty-is [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ valid-sops ..

sublocale initial sb ⊆ valid-reads
proof
  fix i is O R D θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
  show reads-consistent False O m sb
  using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ valid-history
proof
  fix i is O R D θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
  show program.history-consistent program-step θ (hd-prog p sb) sb
  using empty-sb [OF i-bound ts-i] by (auto simp add: program.history-consistent.simps)
qed

sublocale initial sb ⊆ valid-data-dependency
proof
  fix i is O R D θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
  show data-dependency-consistent-instrs (dom θ) is
  using empty-is [OF i-bound ts-i] by auto
next
  fix i is O R D θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
  show load-tmps is∩ UNION (fst ' write-sops sb) = {}
  using empty-is [OF i-bound ts-i] empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ load-tmps-fresh
proof
  fix i is O R D θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts!i = (p,is,θ,sb,D,O,R)
  show load-tmps is∩ dom θ = {}
  using empty-is [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ enough-flushes
proof
  fix i is O R D θ sb p

398
assume i-bound: i < length ts
assume ts-i: tsli = (p, is, θ, sb, D, O, R)
show outstanding-refs is-volatile-Write \( \text{sb} = \{ \} \)
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial_{\text{sb}} \subseteq valid-program-history

proof
fix i is O \( R \) D \( \theta \) sb p sb_1 sb_2
assume i-bound: i < length ts
assume ts-i: tsli = (p, is, θ, sb, D, O, R)
assume sb: sb = sb_1 @ sb_2
show \( \exists \text{isa, instrs sb_2} \hat{\in} \text{isa} \oplus \text{prog-instrs sb_2} \)
using empty-sb [OF i-bound ts-i] empty-is [OF i-bound ts-i] sb by auto
next
fix i is O \( R \) D \( \theta \) sb p
assume i-bound: i < length ts
assume ts-i: tsli = (p, is, θ, sb, D, O, R)
show last-prog p sb = p
using empty-sb [OF i-bound ts-i] by auto
qed

inductive
sim-config:: (p, p store-buffer, bool, owns, rels) thread-config list \times memory \times shared \Rightarrow
(p, unit, bool, owns, rels) thread-config list \times memory \times shared \Rightarrow bool
(- ~ - [60, 60] 100)

where
\[
m = \text{flush-all-until-volatile-write ts_{\text{ab}} m_{\text{ab}}};
\]
\( S = \text{share-all-until-volatile-write ts_{\text{ab}} S_{\text{ab}}};\)
length ts_{\text{ab}} = length ts;
\( \forall i < \text{length ts}_{\text{ab}}.\)

let (p, is_{\text{ab}}, θ, sb, D_{\text{ab}}, O, R) = ts_{\text{ab}}li;
suspends = dropWhile (Not \circ is-volatile-Write_{\text{ab}}) sb
in \( \exists \text{is D instrs suspends } \oplus \text{is}_{\text{ab}} = \text{is} \oplus \text{prog-instrs suspends} \wedge\)
\( D_{\text{ab}} = (D \lor \text{outstanding-refs is-volatile-Write}_{\text{ab}} \text{sb} \neq \{ \}) \wedge\)
tsli = (hd-prog p suspends, θ ∧ (dom θ − read-tmps suspends), ()),
\( D,\)
acquired True (takeWhile (Not \circ is-volatile-Write_{\text{ab}}) sb) O,
release (takeWhile (Not \circ is-volatile-Write_{\text{ab}}) sb) (dom S_{\text{ab}} R )
\]}
\Rightarrow
(ts_{\text{ab}}, m_{\text{ab}}, S_{\text{ab}}) \sim (ts, m, S)
The machine without history only stores writes in the store-buffer.inductive
sim-history-config::
(p, p store-buffer, ‘dirty’, ‘owns’, ‘rels) thread-config list \Rightarrow (p, p store-buffer, bool, owns, rels) thread-config list
\Rightarrow bool
(- ~_{\text{h}} - [60, 60] 100)

where
\[
\forall i < \text{length ts}_{\text{h}};
\]
let (p, is, θ, sb, D, O, R) = ts_{\text{h}}li in
\( ts_{\text{h}}li = (p, is, θ, \text{filter is-Write}_{\text{ab}} sb, D', O', R') \wedge\)
\( (\text{filter is-Write}_{\text{ab}} sb = \{ \} \rightarrow \text{sb}=[\{\}])\)
\]}
\Rightarrow
\text{ts} \sim_{\text{h}} \text{ts}_{\text{h}}
lemma (in initial_{ub}) history-refl: ts \sim_\theta ts
apply -
apply (rule sim-history-config,intros)
apply simp
apply clarsimp
subgoal for i
apply (case-tac ts!i)
apply (drule-tac i=i in empty-sb)
apply assumption
apply auto
done
done

lemma share-all-empty: \forall i p is xs sb D O R. i < length ts \rightarrow ts!i=(p,is,xs,sb,D,O,R) \rightarrow sb=[]
apply (induct ts)
apply clarsimp
apply clarsimp
apply (frule-tac x=0 in spec)
apply clarsimp
apply force
done
done

lemma flush-all-empty: \forall i p is xs sb D O R. i < length ts \rightarrow ts!i=(p,is,xs,sb,D,O,R) \rightarrow sb=[]
\rightarrow flush-all-until-volatile-write ts m = m
apply (induct ts)
apply clarsimp
apply clarsimp
apply (frule-tac x=0 in spec)
apply clarsimp
apply force
done
done

lemma sim-config-emptyE:
assumes empty:
\forall i p is xs sb D O R. i < length ts_{sb} \rightarrow ts_{sb}!i=(p,is,xs,sb,D,O,R) \rightarrow sb=[]
assumes sim: (ts_{sb},m_{sb},S_{sb}) \sim (ts,m,S)
shows S = S_{sb} \land m = m_{sb} \land length ts = length ts_{sb} \land
(\forall i < length ts_{sb}.
let (p, is, @, sb, D, O, R) = ts_{sb}!i
in ts!i = (p, is, @, (), D, O, R))
proof -
from sim
show \?
thesis
apply cases
apply (clarsimp simp add: flush-all-empty [OF empty] share-all-empty [OF empty])
subgoal for i
apply (drule-tac x=i in spec)
apply (cut-tac i=i in empty [rule-format])
apply clarsimp
apply assumption
apply (auto simp add: Let-def)
done
done
qed

lemma sim-config-emptyI:
assumes empty:
\forall i p is xs sb D O R. i < length ts_{sb} \rightarrow ts_{sb}!i=(p,is,xs,sb,D,O,R) \rightarrow sb=[]
assumes leq: length ts = length ts_{sb}
assumes ts: (\forall i < length ts_{sb}.}
let \((p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R}) = \text{ts}_{\text{sb}}\)

\(\text{in ts}! = (p, \text{is}, \emptyset, (), \mathcal{D}, \mathcal{O}, \mathcal{R})\)

shows \((\text{ts}_{\text{sb}, \text{m}_{\text{sb}}, \mathcal{S}_{\text{sb}}}) \sim (\text{ts}_{\text{m}_{\text{sb}}, \mathcal{S}_{\text{sb}}})\)

apply (rule sim-config, intros)

apply (simp add: flush-all-empty [OF empty])

apply (simp add: share-all-empty [OF empty])

apply (simp add: leq)

apply (clarsimp)

apply (frule (1) empty [rule-format])

using \(\text{ts}\)

apply (auto simp add: Let-def)

done

lemma mem-eq-un-eq: \[
\begin{array}{c}
\text{length } \text{ts}' = \text{length } \text{ts}; \\
\forall i < \text{length } \text{ts}'. P (\text{ts}'!i) = Q (\text{ts}!i)
\end{array}
\]

\(\rightarrow (\bigcup_{x \in \text{set } \text{ts}'.} P x) = (\bigcup_{x \in \text{set } \text{ts}.} Q x)\)

apply (auto simp add: in-set-conv-nth)

apply (force dest!: nth-mem)

subgoal for \(x \ i\)

apply (drule-tac \(x=i\) in spec)

apply auto

done

done

lemma (in program) trace-to-steps:

assumes trace: trace c 0 k

shows steps: \(c 0 \Rightarrow_{a^*} c k\)

using trace

proof (induct k)

\(\text{case } 0\)

\(\text{show } c 0 \Rightarrow_{a^*} c 0\)

by auto

next

\(\text{case (Suc } k)\)

have prem: trace c 0 (Suc k) by fact

hence trace c 0 k

by (auto simp add: program-trace-def)

from Suc.hyps [OF this]

have c 0 \(\Rightarrow_{a^*} c k\).

also

term program-trace

from prem interpret program-trace program-step c 0 Suc k.

from step [of k] have c (k) \(\Rightarrow_{a^*} c (\text{Suc } k)\)

by auto

finally show \(?\text{case }\).

qed

lemma (in program) safe-reach-to-safe-reach-upto:

assumes safe-reach: safe-reach-direct safe c_0

shows safe-reach-upto n safe c_0

proof

fix k c l

assume k-n: k \(\leq\) n

assume trace: trace c 0 k

assume c-0: c 0 = c_0

assume l-k: l \(\leq\) k

show safe (c l)

proof

from trace k-n l-k have trace': trace c 0 l

by (auto simp add: program-trace-def)

from trace-to-steps [OF trace']
have $c \Rightarrow_{d^*} c$.

with safe-reach c-0 show safe (c l)
by (cases c l) (auto simp add: safe-reach-def)
qed

lemma (in program-progress) safe-free-flowing-implies-safe-delayed':
assumes init: $\text{initial}_{ab} t_{sb} S_{sb}$
assumes sim: $(t_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)$
assumes safe-reach-ff: safe-reach-direct safe-free-flowing $(ts, m, S)$
shows safe-reach-direct safe-delayed $(ts, m, S)$

proof –
from init
interpret ini: initial$_{ab} ts_{sb} S_{sb}$.
from sim obtain
$m: m = \text{flush-all-until-volatile-write} ts_{sb} m_{sb}$ and
$S: S = \text{share-all-until-volatile-write} ts_{sb} S_{sb}$ and
leq: length $ts_{sb} = \text{length} ts$ and
t-sim: $\forall i < \text{length} ts_{sb}$,
let $(p, i; k, \theta; sb, D_{ab}, O, \mathcal{R}) = ts_{sb}|i$:
suspends = dropWhile (Not o is-volatile-Write$_{ab}$) sb
in $\exists is D. \text{instrs} suspends \theta is = \theta \text{ prog-instrs} suspends \wedge$
$D_{ab} = (D \vee \text{outstanding-refs} \text{is-volatile-Write}_{ab} \text{sb} \neq \{\}) \wedge$
ts$|i = (hd-prog p suspends, is,
$\theta' \cdot (\text{dom} \theta - \text{read-tmps} suspends),(),,$
$D,)$
acquired True (takeWhile (Not o is-volatile-Write$_{ab}$) sb) $O,$
release (takeWhile (Not o is-volatile-Write$_{ab}$) sb) (dom $S_{sb}$ $\mathcal{R}$)
by cases auto

from ini.empty-sb
have shared-eq: $S = S_{sb}$
apply (simp only: $S$)
apply (rule share-all-empty)
apply force
done

have sd: simple-ownership-distinct ts
proof
fix $i, j, p, is, O, \mathcal{R}, D, \theta, j, p, is, O, \mathcal{R}, D, j, \theta, sb, j$
assume i-bound: $i < \text{length} ts$ and
j-bound: $j < \text{length} ts$ and
neq-i-j: $i \neq j$ and

t-sim: $ts|i = (p, i; is, \theta, sb, D_{i}, O_{i}, \mathcal{R}_{i})$ and
\ts-j: $ts|j = (p, j; is, \theta, sb, D_{j}, O_{j}, \mathcal{R}_{j})$
show $(O_{i}) \cap (O_{j}) = \{\}$
proof –
from t-sim [simplified leq, rule-format, OF i-bound] ini.empty-sb [simplified leq, OF i-bound]
have ts-i: $ts_{sb}|i = (p, i; is, \theta, [\_], D_{i}, O_{i}, \mathcal{R}_{i})$
using ts-i
by (force simp add: Let-def)
from t-sim [simplified leq, rule-format, OF j-bound] ini.empty-sb [simplified leq, OF j-bound]
have ts-j: $ts_{sb}|j = (p, j; is, \theta, [\_], D_{j}, O_{j}, \mathcal{R}_{j})$
using ts-j
by (force simp add: Let-def)
from ini.simple-ownership-distinct [simplified leq, OF i-bound j-bound neq-i-j ts-i ts-j]
show ?thesis.
qed

qed

have ro: read-only-unowned $S$ ts
proof
fix i is $O \n R \n D \n \emptyset \n sb \n p$
assume i-bound: $i < \text{length } ts$
assume ts-i: $ts[i] = (p, is, \emptyset, sb, D, O, R)$
show $O \cap \text{read-only } S = \{\}$

proof
from t-sim [simplified leq, rule-format, OF i-bound] ini.empty-sb [simplified leq, OF i-bound]
have ts-i: $ts[i] = (p, is, \emptyset, [\ ], D, O, R)$
using ts-i
by (force simp add: Let-def)
from ini.read-only-unowned [simplified leq, OF i-bound ts-i] shared-eq
show ?thesis by simp
qed

have us: unowned-shared $S \ n ts$
proof
show $-(\bigcup((\lambda(-, -, -, -, O, -). O) \ n \set ts)) \subseteq \text{dom } S$
proof
have $((\bigcup((\lambda(-, -, -, -, O, -). O) \ n \set ts[i])) = ((\bigcup((\lambda(-, -, -, -, O, -). O) \ n \set ts))$
apply clarsimp
apply (rule mem-eq-un-eq)
apply (clarsimp simp add: leq)
apply clarsimp
apply (frule t-sim [rule-format])
apply (clarsimp simp add: Let-def)
apply (drule (1) ini.empty-sb)
apply auto
done
with ini.unowned-shared show ?thesis by (simp only: shared-eq)
qed

\{ 
fix i is $O \n R \n D \n \emptyset \n sb \n p$
assume i-bound: $i < \text{length } ts$
assume ts-i: $ts[i] = (p, is, \emptyset, sb, D, O, R)$
have $R = \text{Map.empty}$
proof
from t-sim [simplified leq, rule-format, OF i-bound] ini.empty-sb [simplified leq, OF i-bound]
have ts-i: $ts[i] = (p, is, \emptyset, [\ ], D, O, R)$
using ts-i
by (force simp add: Let-def)
from ini.empty-rels [simplified leq, OF i-bound ts-i]
show ?thesis .
qed
\}

with us have initial: initial $(ts, m, S)$
by (fastforce simp add: initial-def)

\{ 
fix $ts' \ n S' \ n m'$
assume steps: $(ts, m, S) \Rightarrow_d^+ (ts', m', S')$
have safe-delayed $(ts', m', S')$
proof
from steps-to-trace [OF steps] obtain c k
where trace: $trace c 0 k$ and c-0: $c 0 = (ts, m, S)$ and c-k: $c k = (ts', m', S')$
by auto
from safe-reach-to-safe-reach-upto [OF safe-reach-ff]
have safe-upto-k: safe-reach-upto k safe-free-flowing $(ts, m, S)$,
from safe-free-flowing-implies-safe-delayed [OF - - - - safe-upto-k, simplified, OF initial sd ro us]
have safe-reach-upto k safe-delayed $(ts, m, S)$,
then interpret program-safe-reach-upto program-step k safe-delayed $(ts, m, S)$.
from safe-config [where c=c and k=k and l=k, OF - trace c-0] c-k show ?thesis by simp
qed
}
then show \( ?\thesis \)
  by (clarsimp simp add: safe-reach-def)
qed

lemma map-onws-sb-owned: \( \forall j. \ j < \text{length ts} \implies \text{map O-sb } \text{ts} \! j = (O\_j, sb\_j) \implies \text{map owned ts} \! j = O\_j \)
apply (induct ts)
apply simp
subgoal for t ts j
apply (case-tac j)
apply (case-tac t)
apply auto
done

lemma map-onws-sb-owned': \( \forall j. \ j < \text{length ts} \implies O\_\text{sb} (\text{ts} \! j) = (O\_j, sb\_j) \implies \text{owned (ts} \! j) = O\_j \)
apply (induct ts)
apply simp
subgoal for t ts j
apply (case-tac j)
apply (case-tac t)
apply auto
done
done

lemma read-only-read-acquired-unforwarded-acquire-witness:
\( \forall S O X. \left[ \begin{array}{l}
\text{non-volatile-owned-or-read-only True } S O \text{ sb;}
\text{sharing-consistent } S O \text{ sb; } a \notin \text{read-only } S; a \notin O;
\text{a } \in \text{unforwarded-non-volatile-reads sb X}
\end{array} \right] \implies \right.
\begin{array}{l}
\exists a' \text{ sop } v \text{ A L R W. }
y = y' \text{ } @ \text{Write}_{a'} \text{ True } a' \text{ sop } a \text{ L R W } # \text{ zs } \wedge
\text{a } \in A \wedge a \notin \text{outstanding-refs is-Write}_{a'} \text{ ys } \wedge a' \neq a \vee
\text{a } \in A \wedge a \notin \text{outstanding-refs is-WRITE}_{a'} \text{ ys } \wedge a' \neq a
\end{array}
\left. \right) \wedge
\text{(exists A L R W zs. sb } = y' \text{ } @ \text{Ghost}_{a'} A L R W# \text{ zs } \wedge a \in A \wedge a \notin \text{outstanding-refs is-WRITE}_{a'} \text{ ys})

proof (induct sb)
  case Nil thus \( ?\thesis \) by simp
next
  case (Cons x sb)
  show \( ?\thesis \)
  proof (cases x)
    case (Write_{a'} volatile a' sop v A L R W)
    show \( ?\thesis \)
    proof (cases volatile)
      case True
      note volatile=\this
      from Cons.prems obtain
        nvo': \text{non-volatile-owned-or-read-only True } (S \otimes W R \otimes A L) (O \cup A - R) \text{ sb and}
        a-nro: a \notin \text{read-only } S \text{ and}
        a-unowned: a \notin O \text{ and}
        A-shared-owns: A \subseteq \text{dom } S \cup O \text{ and L-A: L } \subseteq A \text{ and A-R: } A \cap R = \{\} \text{ and}
        R-owns: R \subseteq O \text{ and}
        consis': \text{sharing-consistent } (S \otimes W R \otimes A L) (O \cup A - R) \text{ sb and}
        a-unforw: a \in \text{unforwarded-non-volatile-reads sb (insert a'} X)
      by (clarsimp simp add: Write_{a'} True)
      from unforwarded-not-written [OF a-unforw]
      have a-notin: a \notin \text{insert a'} X.
      hence a'-a: a' \neq a
      by simp
from R-owns a-unowned
have a-R: a \notin R
by auto

show \?thesis
proof (cases a \in A)
case True
then show \?thesis
apply \_
apply (rule disjI1)
apply (rule-tac x=sop in exl)
apply (rule-tac x=a' in exl)
apply (rule-tac x=v in exl)
apply (rule-tac x=\[] in exl)
apply (rule-tac x=\sb in exl)
apply (simp add: Write_{ab} volatile True a'\cdot a)
done

next
case False
with a-unowned R-owns a-nro L-A A-R
obtain a-nro': a \notin read-only (S \oplus W R \ominus A L) and a-unowned': a \notin O \cup A - R
by (force simp add: in-read-only-convs)

from Cons.hyps [OF nvo' consis' a-nro' a-unowned' a-unforw]
have (sop a' v ys zs A L R W.
sb = ys @ Write_{ab} True a' sop v A L R W \# zs \land
a \in A \land a \notin outstanding-refs is-Write_{ab} ys \land a' \neq a) \lor
(\exists A L R W ys zs. sb = ys @ Ghost_{ab} A L R W# zs \land a \in A \land a \notin outstanding-refs is-Write_{ab} ys)
(is ?write \lor ?ghst)

by simp
then show \?thesis
proof
assume ?write

then obtain sop' a'' v' ys zs A \land R' W' where
sb: sb = ys @ Write_{ab} True a'' sop' v' A \land R' W' \# zs and
props: a \in A \land a \notin outstanding-refs is-Write_{ab} ys \land a'' \neq a
by auto

show \?thesis
using props False a-notin sb
apply \_
apply (rule disjI1)
apply (rule-tac x=sop' in exl)
apply (rule-tac x=a'' in exl)
apply (rule-tac x=v' in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Write_{ab} volatile False a'\cdot a)
done

next
assume ?ghst
then obtain ys zs A \land R' W' where
sb: sb = ys @ Ghost_{ab} A \land R' W' \# zs and
props: a \in A \land a \notin outstanding-refs is-Write_{ab} ys
by auto

show \?thesis
using props False a-notin sb
apply \_

apply (rule disjI2)
apply (rule-tac x=A' in exl)
apply (rule-tac x=L' in exl)
apply (rule-tac x=R' in exl)
apply (rule-tac x=W' in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Write sb volatile False a' -a)
done

qed

next
case False from Cons.prems obtain
consis': sharing-consistent S O sb and
a-nro': a \notin read-only S and
a-unowned: a \notin O and
a-ro': a' \in O and
nvo': non-volatile-owned-or-read-only True S O sb and
a-unforw': a \in unforwarded-non-volatile-reads sb (insert a' X)
by (auto simp add: Write sb False split: if-split-asm)

from unforwarded-not-written [OF a-unforw']
have a-notin: a \notin insert a' X.

from Cons.hyps [OF nvo' consis' a-nro' a-unowned a-unforw']
have \( \exists \, s\, o\, p \, a^{'} \, v \, y\, s \, z\, s \, A \, L \, R \, W. \)
  \( \text{sb} = y\, s \, @ \, \text{Write}_{sb} \, \text{True a'} \, s\, o\, p \, v \, A \, L \, R \, W \, \# \, z\, s \, \land \, a \, \in \, A \, \land \, a \, \notin \, \text{outstanding-refs is-Write}_{sb} \, y\, s \, \land \, a' \, \neq \, a \) \lor
  \( (\exists \, A \, L \, R \, W \, y\, s \, \text{sb} = y\, s \, @ \, \text{Ghost}_{sb} \, A \, L \, R \, W \, \# \, z\, s \, \land \, a \, \in \, A \, \land \, a \, \notin \, \text{outstanding-refs is-Write}_{sb} \, y\, s) \)
(is ?write \lor \?ghost)
by simp
then show \?thesis
proof
assume \?write
then obtain \( \text{sop}^{'} \, a^{''} \, v' \, y\, s \, z\, s \, A' \, L' \, R' \, W' \, \) where
  \( \text{sb: sb} = y\, s \, @ \, \text{Write}_{sb} \, \text{True a'} \, s\, o\, p \, v' \, A' \, L' \, R' \, W' \, \# \, z\, s \, \land \, a \, \in \, A' \, a \, \notin \, \text{outstanding-refs is-Write}_{sb} \, y\, s \, \land \, a'' \, \neq \, a \)
by auto

show \?thesis
using props False a-notin sb
apply –
apply (rule disjI1)
apply (rule-tac x=sop' in exl)
apply (rule-tac x=a' in exl)
apply (rule-tac x=v' in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Write sb False )
done

next
assume \?ghost
then obtain y\, s \, z\, s \, A' \, L' \, R' \, W' \, \) where
  \( \text{sb: sb} = y\, s \, @ \, \text{Ghost}_{sb} \, A' \, L' \, R' \, W' \, \# \, z\, s \, \land \, a \, \in \, A' \, a \, \notin \, \text{outstanding-refs is-Write}_{sb} \, y\, s \)
by auto
show \(?thesis
using props False a-notin sb
apply –
apply (rule disjI2)
apply (rule-tac x=A’ in exl)
apply (rule-tac x=L’ in exl)
apply (rule-tac x=R’ in exl)
apply (rule-tac x=W’ in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Write sb False )
done
qed

then show \(?thesis
proof
assume \(?write

then obtain sop’ a’’ v’ ys zs A’ L’ R’ W’ where
sb: sb = ys @ Write sb True a’’ sop v A’ L’ R’ W’ # zs ∧
a’’ a’ ∈ A ∧ a’ /∈ outstanding-refs is-Write sb ys ∧ a’’ ≠ a) ∨
(∃ A’ L’ R’ W’ ys zs. sb = ys @ Ghost sb A’ L’ R’ W’ # zs ∧ a ∈ A ∧ a’ /∈ outstanding-refs is-Write sb)
(is ?write ∨ ?ghst)
by simp
then show \(?thesis

then obtain sop’ a’’ v’ ys zs A’ L’ R’ W’ where
sb: sb = ys @ Write sb True a’’ sop v A’ L’ R’ W’ # zs and
props: a ∈ A’ a /∈ outstanding-refs is-Write sb ys ∧ a’’ ≠ a
by auto

show \(?thesis
using props sb
apply –
apply (rule disjI1)
apply (rule-tac x=sop’ in exl)
apply (rule-tac x=a’’ in exl)
apply (rule-tac x=v’ in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Read sb)
done

next
assume \(?ghst
then obtain ys zs A’ L’ R’ W’ where
sb: sb = ys @ Ghost sb A’ L’ R’ W’ # zs and
props: a ∈ A’ a /∈ outstanding-refs is-Write sb ys
by auto

show \(?thesis
using props sb
apply –
apply (rule disjI2)
apply (rule-tac x=A' in exl)
apply (rule-tac x=L' in exl)
apply (rule-tac x=R' in exl)
apply (rule-tac x=W' in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Read_{sb})
done
qed
next
case Prog_{sb}
from Cons.prems
obtain
  consis': sharing-consistent S O sb and
  a-nro': a \notin \{read-only, outstanding-refs\} and
  a-unowned: a \notin O and
  nvo': non-volatile-owned-or-read-only True S O sb and
  a-unforw: a \in unforwarded-non-volatile-reads sb X
by (auto simp add: Prog_{sb})
from Cons.hyps [OF nvo' consis' a-nro' a-unowned a-unforw]
have \(\exists \text{sop}' \ a' \ v \ ys \ zs \ A L R W.\)
  \(\text{sb} = ys \oplus \text{Write}_{sb} \ True \ a' \ \text{sop} v \ A L R W \ # \ zs \ \land\)
  \(a \in A \land a' \notin \text{outstanding-refs is-Write}_{sb} \ ys \ \land \ a' \neq a) \lor\)
  \(\exists A L R W \ ys \ zs. \ \text{sb} = ys \oplus \text{Ghost}_{sb} A L R W # zs \ \land \ a \in A \land a \notin \text{outstanding-refs is-Write}_{sb} \ ys\)
(is ?write \lor ?ghst)
by simp
then show ?thesis
proof
  assume ?write

  then obtain sop' a'' v' ys zs A' L' R' W' where
  \(\text{sb} = ys \oplus \text{Write}_{sb} \ True \ a' \ \text{sop} v' A' L' R' W' \ # \ zs\ \land\)
  props: a \in A' a \notin \text{outstanding-refs is-Write}_{sb} \ ys \land a'' \neq a
by auto

  show ?thesis
  using props sb
  apply –
  apply (rule disjI1)
  apply (rule-tac x=sop' in exl)
  apply (rule-tac x=a'' in exl)
  apply (rule-tac x=v' in exl)
  apply (rule-tac x=(x#ys) in exl)
  apply (rule-tac x=zs in exl)
  apply (simp add: Prog_{sb})
done
next
assume ?ghst
then obtain ys zs A' L' R' W' where
  \(\text{sb} = ys \oplus \text{Ghost}_{sb} A' L' R' W' \ # \ zs\ \land\)
  props: a \in A' a \notin \text{outstanding-refs is-Write}_{sb} \ ys
by auto

  show ?thesis
  using props sb
  apply –
apply (rule disjI2)
apply (rule-tac x=A' in exl)
apply (rule-tac x=L' in exl)
apply (rule-tac x=R' in exl)
apply (rule-tac x=W' in exl)
apply (rule-tac x=(x#ys) in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Prog_{ab})
done

qed

next
case (Ghost_{ab} A L R W)
from Cons.prems obtain
  nvo': non-volatile-owned-or-read-only True (S ⊕ W R ⊕ A L) (O ∪ A − R) sb and
  a-nro: a /∈ read-only S and
  a-unowned: a /∈ O and
  A-shared-owns: A ⊆ dom S ∪ O and L-A: L ⊆ A and A-R: A ∩ R = {} and
  Rowns: R ⊆ O and
  consis': sharing-consistent (S ⊕ W R ⊕ A L) (O ∪ A − R) sb and
  a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (clarsimp simp add: Ghost_{ab})

show ?thesis
proof (cases a ∈ A)
case True
  then show ?thesis
  apply -
  apply (rule disjI2)
  apply (rule-tac x=A in exl)
  apply (rule-tac x=L in exl)
  apply (rule-tac x=R in exl)
  apply (rule-tac x=W in exl)
  apply (rule-tac x=[] in exl)
  apply (simp add: Ghost_{ab} True)
done
next
case False
with a-unowned a-nro L-A R-owns a-nro L-A-R
obtain a-nro': a /∈ read-only (S ⊕ W R ⊕ A L) and a-unowned': a /∈ O ∪ A − R
by (force simp add: in-read-only-convs)
from Cons.hyps [OF nvo' consis' a-nro' a-unowned' a-unforw]
have (∃ sop a' v ys zs A L R W.
  sb = ys @ Write_{ab} True a' sop v A L R W # zs ∧
  a ∈ A ∧ a /∈ outstanding-refs is-Write_{ab} ys ∧ a' ≠ a) ∨
  (∃ A L R W ys zs. sb = ys @ Ghost_{ab} A L R W# zs ∧ a ∈ A ∧ a /∈ outstanding-refs is-Write_{ab} ys)
(is ?write ∨ ?ghst)
by simp
  then show ?thesis
  proof
  assume ?write
then obtain sop' a'' v' ys zs A' L' R' W' where
  sb: sb = ys @ Write_{ab} True a'' sop' v' A' L' R' W' # zs and
  props: a ∈ A' a /∈ outstanding-refs is-Write_{ab} ys ∧ a'' ≠ a
by auto

show ?thesis
using props sb

409
apply –
apply (rule disjI1)
apply (rule-tac x=sop' in exl)
apply (rule-tac x=a'' in exl)
apply (rule-tac x=v' in exl)
apply (rule-tac x={x#ys} in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Ghost sb False )
done

next
assume ?ghst
then obtain ys zs A' L' R' W' where
  sb: sb = ys @ Ghost sb A' L' R' W' # zs and
  props: a ∈ A' a /∈ outstanding-refs is-Write sb ys
by auto

show ?thesis
using props sb
apply –
apply (rule disjI2)
apply (rule-tac x=A' in exl)
apply (rule-tac x=L' in exl)
apply (rule-tac x=R' in exl)
apply (rule-tac x=W' in exl)
apply (rule-tac x={x#ys} in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Ghost sb False )
done
qed

lemma release-shared-exchange-weak:
assumes shared-eq: ∀ a ∈ O ∪ all-acquired sb. (S':shared) a = S a
assumes consis: weak-sharing-consistent O sb
shows release sb (dom S') R = release sb (dom S) R
using shared-eq consis
proof (induct sb arbitrary: S S' O R)
  case Nil thus ?case by auto
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True
      from Cons.prems obtain
        L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
        consis': weak-sharing-consistent (O ∪ A − R) sb and
        shared-eq: ∀ a ∈ O ∪ A − R ∪ all-acquired sb. S' a = S a
      by (clarsimp simp add: Write sb True )
      from shared-eq
        have shared-eq': ∀ a ∈ O ∪ A − R ∪ all-acquired sb. (S' ⊕_W R ⊕_A L) a = (S ⊕_W R ⊕_A L) a
        by (auto simp add: augment-shared-def restrict-shared-def)
      from Cons.hyps OF shared-eq' consis'
  end
qed

410
have release sb (dom (S' ⊕ W R ⊖ A L)) Map.empty = release sb (dom (S ⊕ W R ⊖ A L)) Map.empty.
then show ?thesis
  by (auto simp add: Write sb True domIff)
next
case False with Cons show ?thesis
by (auto simp add: Write sb)
qed
next
case Read sb with Cons show ?thesis
by auto
next
case Prog sb with Cons show ?thesis
by auto
next
case (Ghost sb A L R W)
from Cons.prems obtain
L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
consis': weak-sharing-consistent (O ∪ A − R) sb and
shared-eq: ∀a ∈ O ∪ A ∪ all-acquired sb. S' a = S a
by (clarsimp simp add: Ghost sb)
from shared-eq
have shared-eq': ∀a ∈ O ∪ A − R ∪ all-acquired sb. (S' ⊕ W R ⊖ A L) a = (S ⊕ W R ⊖ A L) a
by (auto simp add: augment-shared-def restrict-shared-def)

then show ?thesis
by simp

from augment-rels-shared-exchange [OF this]
have (augment-rels (dom S') R R) = (augment-rels (dom S) R R).

lemma read-only-share-all-shared: \(\forall S. \forall a \in \text{read-only} (\text{share sb S})\)
implies a ∈ \text{read-only} S ∪ all-shared sb
proof (induct sb)
  case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
  case (Write sb volatile v A L R W)
  show ?thesis
  proof (cases volatile)
    case True
    with Write sb Cons.hyps [of (S ⊕ W R ⊖ A L)] Cons.prems
    show ?thesis
    by (auto simp add: read-only-def augment-shared-def restrict-shared-def
      split: if-split-asm option.splits)
  next
case False with Write sb Cons show ?thesis by auto
  qed
next
case Read sb with Cons show ?thesis by auto
next
case Prog sb with Cons show ?thesis by auto
next
case (Ghost_A, A L R W)
with Cons.hyps [of (S ⊕ W R ⊕ A L)] Cons.prems

show ?thesis
  by (auto simp add: read-only-def augment-shared-def restrict-shared-def
       split: if-split-asm option.splits)
qed

lemma read-only-shared-all-until-volatile-write-subset':
  \forall S.
  read-only (share-all-until-volatile-write ts S) ⊆
  read-only S ∪ (\bigcup ((\lambda (\cdot, \cdot, \cdot, sb, \cdot, \cdot, \cdot). all-shared (takeWhile (Not o is-volatile-Write_ab) sb)) ' set ts))
proof (induct ts)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  obtain p is ORD \theta sb
    where t: t = (p, is, \theta, sb, D, O, R)
    by (cases t)
  have aargh: (Not o is-volatile-Write_ab) = (\lambda a. ~ is-volatile-Write_ab a)
    by (rule ext) auto
  let ?take-sb = (takeWhile (Not o is-volatile-Write_ab) sb)
  let ?drop-sb = (dropWhile (Not o is-volatile-Write_ab) sb)

  {
    fix a
    assume a-in: a ∈ read-only
      (share-all-until-volatile-write ts
       (share ?take-sb S)) and
    a-notin-shared: a /∈ read-only S and
    a-notin-rest: a /∈ (\bigcup ((\lambda (\cdot, \cdot, \cdot, sb, \cdot, \cdot, \cdot). all-shared (takeWhile (Not o is-volatile-Write_ab) sb)) ' set ts))
    have a ∈ all-shared (takeWhile (Not o is-volatile-Write_ab) sb)
      proof
        from Cons.hyps [of (share ?take-sb S)] a-in a-notin-rest
        have a ∈ read-only (share ?take-sb S)
          by (auto simp add: aargh)
        from read-only-share-all-shared [OF this] a-notin-shared
        show ?thesis by auto
      qed
  }
  then show ?case
    by (auto simp add: t aargh)
qed

lemma read-only-share-acquired-all-shared:
  \forall O S. weak-sharing-consistent O sb \Longrightarrow O ∩ read-only S = \{\} \Longrightarrow
  a ∈ read-only (share sb S) \Longrightarrow a ∈ O ∪ all-acquired sb \Longrightarrow a ∈ all-shared sb
proof (induct sb)
  case Nil thus ?case by auto
next
  case (Cons x sb)
  show ?case
    proof (cases x)

412
case \((\text{Write}_{\text{sb}} \text{ volatile } a \text{ sop } v A L R W)\)

show \(?\text{thesis}\)

proof (cases volatile)

case True

note \(\text{volatile} = \text{this}\)

from Cons.prems obtain

\(\text{owns-ro}: O \cap \text{read-only } S = \{\}\) and \(\text{L-A}: L \subseteq A\) and \(\text{A-R}: A \cap R = \{\}\) and

\(\text{R-owns}: R \subseteq O\) and \(\text{consis}': \text{weak-sharing-consistent } (O \cup A - R)\) \(sb\) and

\(\text{a-share}: a \in \text{read-only } (\text{share } sb (S \oplus W R \ominus A L))\) and

\(\text{a-A-all}: a \in O \cup A \cup \text{all-acquired } sb\)

by (clarsimp simp add: \(\text{Write}_{\text{sb}} \text{ True}\))

from \(\text{owns-ro} A-R \text{ R-owns have owns-ro}': (O \cup A - R) \cap \text{read-only } (S \oplus W R \ominus A L) = \{\}\)

by (auto simp add: in-read-only-cons)

from Cons.hyps [OF \(\text{consis}' \text{ owns-ro}' \text{ a-share}\)]

show \(?\text{thesis}\)

using \(\text{L-A} \text{ \text{A-R} \text{ \text{R-owns} \text{ owns-ro} \text{ a-A-all}}\)

by (auto simp add: \(\text{Write}_{\text{sb}} \text{ volatile} \text{ augment-shared-def} \text{ restrict-shared-def} \text{ read-only-def}\) domlff split: if-split-as)

next

case False

with Cons \(\text{Write}_{\text{sb}}\) show \(?\text{thesis}\) by (auto)

qed

next

case \(\text{Read}_{\text{sb}}\) with Cons show \(?\text{thesis}\) by auto

next

case \(\text{Prog}_{\text{sb}}\) with Cons show \(?\text{thesis}\) by auto

next

case \(\text{Ghost}_{\text{sb}}\ L R W\)

from Cons.prems obtain

\(\text{owns-ro}: O \cap \text{read-only } S = \{\}\) and \(\text{L-A}: L \subseteq A\) and \(\text{A-R}: A \cap R = \{\}\) and

\(\text{R-owns}: R \subseteq O\) and \(\text{consis}': \text{weak-sharing-consistent } (O \cup A - R)\) \(sb\) and

\(\text{a-share}: a \in \text{read-only } (\text{share } sb (S \oplus W R \ominus A L))\) and

\(\text{a-A-all}: a \in O \cup A \cup \text{all-acquired } sb\)

by (clarsimp simp add: \(\text{Ghost}_{\text{sb}}\))

from \(\text{owns-ro} A-R \text{ R-owns have owns-ro}': (O \cup A - R) \cap \text{read-only } (S \oplus W R \ominus A L) = \{\}\)

by (auto simp add: in-read-only-cons)

from Cons.hyps [OF \(\text{consis}' \text{ owns-ro}' \text{ a-share}\)]

show \(?\text{thesis}\)

using \(\text{L-A} \text{ \text{A-R} \text{ \text{R-owns} \text{ owns-ro} \text{ a-A-all}}\)

by (auto simp add: \(\text{Ghost}_{\text{sb}} \text{ \text{augment-shared-def} \text{ \text{restrict-shared-def} \text{ \text{read-only-def}\) domlff split: if-split-as})}

qed

qed

lemma \(\text{read-only-share-unowned}' : (O \ominus S) \cap \text{read-only } S = \{\}\) \(\implies a \in \text{read-only } (\text{share } sb S)\)

proof (induct \(sb\))

case Nil thus \(?\text{case by simp}\)

next

case \(\text{Cons } x\) \(sb\)

show \(?\text{case}\)

proof (cases \(x\))

case \(\text{Write}_{\text{sb}}\ \text{volatile } a \text{ sop } v A L R W\)

show \(?\text{thesis}\)

proof (cases volatile)

case False

with Cons \(\text{Write}_{\text{sb}}\) show \(?\text{thesis}\) by auto

next

next

next

next

next

next

next

next

next

next

413
case True
from Cons.prems obtain
owns-ro: O ∩ read-only S = {\} and L-A: L ⊆ A and A-R: A ∩ R = {\} and
R-owns: R ⊆ O and consis': weak-sharing-consistent (O ∪ A − R) sb and
a-share: a ∈ read-only S and
a-notin: a ∉ O a ∉ A a ∉ all-acquired sb
by (clarsimp simp add: Writeab True)
from owns-ro A-R R-owns have owns-ro': (O ∪ A − R) ∩ read-only (S ⊕ W R ⊕ A L) = {\}
by (auto simp add: in-read-only-convs)
from a-notin have a-notin': a ∉ O ∪ A − R ∪ all-acquired sb
by auto
from a-share a-notin L-A A-R R-owns have a-ro': a ∈ read-only (S ⊕ W R ⊕ A L)
by (clarsimp simp add: read-only-def restrict-shared-def augment-shared-def)
from Cons.hyps [OF consis' owns-ro' a-notin' a-ro']
have a ∈ read-only (share sb (S ⊕ W R ⊕ A L))
by auto
then show ?thesis
by (auto simp add: Writeab True)
qed
next
case Readab with Cons show ?thesis by auto
next
case Progab with Cons show ?thesis by auto
next
case (Ghostab A L R W)
from Cons.prems obtain
owns-ro: O ∩ read-only S = {\} and L-A: L ⊆ A and A-R: A ∩ R = {\} and
R-owns: R ⊆ O and consis': weak-sharing-consistent (O ∪ A − R) sb and
a-share: a ∈ read-only S and
a-notin: a ∉ O a ∉ A a ∉ all-acquired sb
by (clarsimp simp add: Ghostab)
from owns-ro A-R R-owns have owns-ro': (O ∪ A − R) ∩ read-only (S ⊕ W R ⊕ A L) = {\}
by (clarsimp simp add: in-read-only-convs)
from a-notin have a-notin': a ∉ O ∪ A − R ∪ all-acquired sb
by auto
from a-share a-notin L-A A-R R-owns have a-ro': a ∈ read-only (S ⊕ W R ⊕ A L)
by (clarsimp simp add: read-only-def restrict-shared-def augment-shared-def)
from Cons.hyps [OF consis' owns-ro' a-notin' a-ro']
have a ∈ read-only (share sb (S ⊕ W R ⊕ A L))
by auto
then show ?thesis
by (auto simp add: Ghostab)
qed

lemma release-False-mono:
\[ S \cap R. R a = Some False \implies \text{outstanding-refs is-volatile-Writeab sb} = {\} \implies \]
release sb S \cap R a = Some False
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Ghostab A L R W)
have rels-a: R a = Some False by fact
then have (augment-rels S R R) a = Some False
by (clarsimp simp add: augment-rels-def)
from Cons.hyps [where $\mathcal{R} = (\text{augment-rels } S \mathcal{R})$, OF this] Cons.prems
  show ?thesis
  by (clarsimp simp add: Ghost_ab)
next
  case Write_{ab} with Cons show ?thesis by auto
next
  case Read_{ab} with Cons show ?thesis by auto
next
  case Prog_{ab} with Cons show ?thesis by auto
qed
qed

lemma release-False-mono-take:
  $\forall S \mathcal{R}. \mathcal{R} a = \text{Some False} \Longrightarrow \text{release}(\text{takeWhile}(\text{Not} \circ \text{is-volatile-Write}_{ab}) sb) S \mathcal{R} a = \text{Some False}$
proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Ghost_{ab} A L R W)
    have aargh: $(\lambda a. \neg \text{is-volatile-Write}_{ab} a)$ by (rule ext) auto
    show ?thesis
    proof (clarsimp simp add: Ghost_ab aargh)
      from dist L-A A-R R-owns have dist’: read-only $(S \oplus W R \ominus A L) \cap (O \cup A - R) = \{\}$
      by (auto simp add: in-read-only-convs)
    qed
  qed

lemma shared-switch:
  $\forall S O. \begin{array}{l}
  \text{weak-sharing-consistent } O; \text{read-only } S \cap O = \{\} ; \\
  S a \neq \text{Some False}; \text{share } sb S a = \text{Some False} \\
  \Longrightarrow a \in O \cup \text{all-acquired } sb
  \end{array}$
proof (induct sb)
  case Nil thus ?case by (auto simp add: read-only-def)
next
  case (Cons x sb)
  have aargh: $(\lambda a. \neg \text{is-volatile-Write}_{ab} a) = (\lambda a. \neg \text{is-volatile-Write}_{ab} a)$
    by (rule ext) auto
  show ?case
  proof (cases x)
    case (Ghost_{ab} A L R W)
    from Cons.prems obtain
      dist: read-only $S \cap O = \{\}$ and
      share: $S a \neq \text{Some False}$ and
      share’: share sb $(S \oplus W R \ominus A L) a = \text{Some False}$ and
      L-A: $L \subseteq A$ and A-R: $A \cap R = \{\}$ and R-owns: $R \subseteq O$ and
      consis’: weak-sharing-consistent $(O \cup A - R) sb$ by (clarsimp simp add: Ghost_{ab} aargh)
    from dist L-A A-R R-owns have dist’: read-only $(S \oplus W R \ominus A L) \cap (O \cup A - R) = \{\}$
    by (auto simp add: in-read-only-convs)
  qed
proof (cases \((S \oplus W R \ominus A L) a = \text{Some False})
  
  case False
  
  from Cons.hyps [OF consis' dist' this share]
  
  show \(\text{thesis}\) by (auto simp add: Ghost\textsubscript{sb})
  
next
  case True
  
  with share L-A A-R R-owns dist
  
  have a \(\in\) O \(\cup\) A
  
  by (cases \(S a\))
    
    (auto simp add: augment-shared-def restrict-shared-def read-only-def split: if-split-asm)
    
  thus \(\text{thesis}\) by (auto simp add: Ghost\textsubscript{sb})
  
qed

next
  case (Write\textsubscript{sb} volatile a' sop v A L R W)
  
  show \(\text{thesis}\)
  
proof (cases volatile)
  
  case True
  
  note volatile = this
  
  from Cons.prems obtain
  
  dist: read-only \(S \cap O = \{\}\) and
  
  share: \(S a \neq \text{Some False}\) and
  
  share': share sb \((S \oplus W R \ominus A L) a = \text{Some False}\) and
  
  L-A: L \(\subseteq\) A and A-R: A \(\cap\) R = \(\{\}\) and R-owns: R \(\subseteq\) O and
  
  consis': weak-sharing-consistent \((O \cup A - R)\) sb by (clarsimp simp add: Write\textsubscript{sb} True aargh)
  
  from dist L-A A-R R-owns have dist': read-only \((S \oplus W R \ominus A L) \cap (O \cup A - R) = \{\}\)
  
  by (auto simp add: in-read-only-convs)
    
  show \(\text{thesis}\)
  
proof (cases \((S \oplus W R \ominus A L) a = \text{Some False})
  
  case False
  
  with share L-A A-R R-owns dist
  
  have a \(\in\) O \(\cup\) A
  
  by (cases \(S a\))
    
    (auto simp add: augment-shared-def restrict-shared-def read-only-def split: if-split-asm)
    
  thus \(\text{thesis}\) by (auto simp add: Write\textsubscript{sb} volatile)
  
qed

next
  case False
  
  with Cons show \(\text{thesis}\) by (auto simp add: Write\textsubscript{sb})
  
qed

next
  case Read\textsubscript{sb} with Cons show \(\text{thesis}\) by (auto)

next
  case Prog\textsubscript{sb} with Cons show \(\text{thesis}\) by (auto)

qed

qed

lemma shared-switch-release-False:

\[ \forall S R. \]

outstanding-refs is-volatile-Write\textsubscript{sb} sb = \{\};

a \(\notin\) dom \(S\);

a \(\in\) dom (share sb \(S\)])

\implies

release sb (dom \(S\)) R a = Some False

proof (induct sb)

  case Nil thus \(\text{case by (auto simp add: read-only-def})

416
next
    case (Cons x sb)
    have aargh: \((\text{Not} \circ \text{is-volatile-Write}_{ab}) = (\lambda a. \neg \text{is-volatile-Write}_{ab} \ a)\)
        by (rule ext) auto
    show ?case
    proof (cases x)
    case (Ghost_{ab} A L R W)
    from Cons.prems obtain
        a-notin: \(a \notin \text{dom } S\) and
        share: \(a \in \text{dom } (\text{share sb } (S \oplus W R \ominus A L)\) and
        out': outstanding-refs is-volatile-Write_{ab} sb = {}
        by (clarsimp simp add: Ghost_{ab} aargh)
    show ?thesis
    proof (clarsimp)
    case (Ghost sb A L R W)
    from Cons.prems obtain
        a-notin: \(a \notin \text{dom } S\) and
        share: \(a \in \text{dom } (\text{share sb } (S \oplus W R \ominus A L)\) and
        out': outstanding-refs is-volatile-Write_{ab} sb = {}
        by (clarsimp simp add: Ghost_{ab} aargh)
    qed
    next
    case Write_{ab} with Cons
    show ?thesis by (clarsimp split: if-split-asm)
    next
    case Read_{ab} with Cons
    show ?thesis by auto
    next
    case Prog_{ab} with Cons
    show ?thesis by auto
    qed
    qed

lemma release-not-unshared-no-write:
\(\forall S \ R. \ [\]
    \text{outstanding-refs is-volatile-Write}_{ab} sb = \{\};
\text{non-volatile-writes-unshared } S\ sb;
\text{release sb } (\text{dom } S) R \ a \neq \text{Some False};
\ a \in \text{dom } (\text{share sb } S)\]
\implies a \notin \text{outstanding-refs is-non-volatile-Write}_{ab} sb

proof (induct sb)
    case Nil thus ?case by (auto simp add: read-only-def)
next
    case (Cons x sb)
    have aargh: \((\text{Not} \circ \text{is-volatile-Write}_{ab}) = (\lambda a. \neg \text{is-volatile-Write}_{ab} \ a)\)
        by (rule ext) auto
    show ?case
    proof (cases x)
    case (Ghost_{ab} A L R W)
    from Cons.prems obtain
        share: \(a \in \text{dom } (\text{share sb } (S \oplus W R \ominus A L)\) and
        rel: \(\text{release sb}
        (\text{dom } (S \oplus W R \ominus A L)) \text{ (augment-rels } (\text{dom } S) R R) \ a \neq \text{Some False} \text{ and}
        out': outstanding-refs is-volatile-Write_{ab} sb = {} \text{ and}
    qed

417
nvu: non-volatile-writes-unshared \((S \oplus_{W} R \ominus_{A} L)\) sb
by (clarsimp simp add: Ghost sb)

from Cons.hyps [OF out' nvu rel share]
show ?thesis by (auto simp add: Ghost sb)
next
case (Write_{ab} volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case True with Write_{ab} Cons.prems have False by auto
thus ?thesis ..
next
case False
note not-vol = this
from Cons.prems obtain
  rel: release sb (dom S) R a \neq Some False and
out': outstanding.refs is-volatile-Write_{ab} sb = {} and
nvo: non-volatile-writes-unshared S sb and
a'-not-dom: a' \notin dom S and
a-dom: a \in dom (share sb S)
by (auto simp add: Write_{ab} False)
from Cons.hyps [OF out' nvo rel a-dom]
have a-notin-rest: a \notin outstanding.refs is-non-volatile-Write_{ab} sb.

show ?thesis
proof (cases a'=a)
case False with a-notin-rest
show ?thesis by (clarsimp simp add: Write_{ab} not-vol)
next
case True
from shared-switch-release-False [OF out' a'-not-dom [simplified True] a-dom]
have release sb (dom S) R a = Some False.
  with rel have False by simp
thus ?thesis ..
qed
qed
next
case Read_{ab} with Cons show ?thesis by auto
next
case Prog_{ab} with Cons show ?thesis by auto
qed
qed

corollary release-not-unshared-no-write-take:
assumes nvw: non-volatile-writes-unshared S (takeWhile (Not \circ is-volatile-Write_{ab}) sb)
assumes rel: release (takeWhile (Not \circ is-volatile-Write_{ab}) sb) (dom S) R a \neq Some False
assumes a-in: a \in dom (share (takeWhile (Not \circ is-volatile-Write_{ab}) sb) S)
shows
  a \notin outstanding.refs is-non-volatile-Write_{ab} (takeWhile (Not \circ is-volatile-Write_{ab}) sb)
using release-not-unshared-no-write [OF takeWhile-not-vol-write-outstanding.refs [of sb] nvw rel a-in]
by simp

lemma read-only-unacquired-share':
\(\forall S. O. \ (O \cap \text{read-only } S = \{\}); \ \text{weak-sharing-consistent } O \ sb \; a \in \text{read-only } S;\]
a \notin all-shared sb; a \notin acquired True sb O \]
\implies a \in \text{read-only } (share sb S)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
  case (Write sb volatile a’ sop v A L R W)
  show ?thesis
  proof (cases volatile)
    case True
    note volatile=this
  from Cons.prems
  obtain a-ro: a ∈ read-only S and a-R: a ∉ R and a-unsh: a ∉ all-shared sb and
  owns-ro: O ∩ read-only S = {} and
  L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
  consis’: weak-sharing-consistent (O ∪ A − R) sb and
  a-notin: a ∉ acquired True sb (O ∪ A − R)
  by (clarsimp simp add: Write sb True)
  show ?thesis
  proof (cases a ∈ A)
    case True
    with a-R have a ∈ O ∪ A − R by auto
    from all-shared-acquired-in [OF this a-unsh]
    have a ∈ acquired True sb (O ∪ A − R) by auto
    with a-notin have False by auto
    thus ?thesis ..
  next
  case False
  from owns-ro A-R R-owns have owns-rot: (O ∪ A − R) ∩ read-only (S ⊕w R ⊕a L) = {}
  by (auto simp add: in-read-only-convs)
  from a-ro False own-rot R-owns L-A have a-ro’: a ∈ read-only (S ⊕w R ⊕a L)
  by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF owns-rot’ conis’ a-ro’ a-unsh a-notin]
  show ?thesis
  by (clarsimp simp add: Write sb False)
  qed
  next
  case False
  with Cons show ?thesis
  by (clarsimp simp add: Write sb False)
  qed
  next
  case Read sb with Cons show ?thesis by (clarsimp)
  next
  case Prog sb with Cons show ?thesis by (clarsimp)
  next
  case (Ghost sb A L R W)
  from Cons.prems
  obtain a-ro: a ∈ read-only S and a-R: a ∉ R and a-unsh: a ∉ all-shared sb and
  owns-ro: O ∩ read-only S = {} and
  L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
  consis’: weak-sharing-consistent (O ∪ A − R) sb and
  a-notin: a ∉ acquired True sb (O ∪ A − R)
  by (clarsimp simp add: Ghost sb)
  show ?thesis
  proof (cases a ∈ A)
    case True
    with a-R have a ∈ O ∪ A − R by auto
    from all-shared-acquired-in [OF this a-unsh]
    have a ∈ acquired True sb (O ∪ A − R) by auto
    with a-notin have False by auto
    thus ?thesis ..
  qed

419
next
  case False
  from owns-ro A-Rindy owns-ro have owns-ro': (O ∪ A − R) ∩ read-only (S ⊕ W R ⊕ A L) = {}
    by (auto simp add: in-read-only-convs)
  from a-ro False owns-ro R-owns
  have a-ro': a ∈ read-only (S ⊕ W R ⊖ A L)
    by (auto simp add: in-read-only-convs)
  from Cons.hyps [OF owns-ro′ consis′ a-ro′ a-unsh a-notin]
  show ?thesis
    by (clarsimp simp add: Ghost[simp])
  qed
qed

lemma read-only-share-all-until-volatile-write-unacquired':
  A S. [ownership-distinct ts; read-only-unowned S ts; weak-sharing-consis ts; ∀ i < length ts.
    let (−,−,−,sb,O,R) = ts!i in
      a /∈ acquired True (takeWhile (Not ◦ is-volatile-Write sb) O ∧
      a /∈ all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb)
    );
    a ∈ read-only S]
  ⇒ a ∈ read-only (share-all-until-volatile-write ts S)
proof (induct ts)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  obtain p is O R D @ sb where
    t = (p,is,@,sb,D,O,R)
    by (cases t)
  have dist: ownership-distinct (t#ts) by fact
  then interpret ownership-distinct t#ts .
  from ownership-distinct-tl [OF dist]
  have dist': ownership-distinct ts.

  have aargh: (Not o is-volatile-Write sb) = (λa. ¬ is-volatile-Write sb a)
    by (rule ext) auto
  have a-ro: a ∈ read-only S by fact
  have ro-unowned: read-only-unowned S (t#ts) by fact
  then interpret read-only-unowned S t#ts .
  have consis: weak-sharing-consis (t#ts) by fact
  then interpret weak-sharing-consis t#ts .

  note consis' = weak-sharing-consis-tl [OF consis]

  let ?take-sb = (takeWhile (Not o is-volatile-Write sb) sb)
  let ?drop-sb = (dropWhile (Not o is-volatile-Write sb) sb)

  from weak-sharing-consis [of 0] t
  have consis-sb: weak-sharing-consistent O sb
    by force
  with weak-sharing-consistent-append [of O ?take-sb ?drop-sb]
  have consis-take: weak-sharing-consistent O ?take-sb sb
    by auto

  have ro-unowned': read-only-unowned (share ?take-sb S) ts
proof
show \( \bigcap \) read-only\((\text{share } ?\text{take-sb} S)\) = \{

proof

−

\begin{align*}
\text{fix} \ a \\
\text{assume} \ a\text{-owns: } & a \in O_j \\
\text{assume} \ a\text{-ro: } & a \in \text{read-only}\ (\text{share } ?\text{take-sb} S) \\
\text{have } & \text{False} \\
\text{proof} & \text{−}
\end{align*}

−

\begin{align*}
\text{from ownership-distinct [of } 0 \text{ Suc } j] \ j\text{-bound } j\text{th } t \\
\text{have dist: } & (O \cup \text{all-acquired sb}) \cap (O_j \cup \text{all-acquired sb}_j) = \{
\text{by } \text{fastforce}
\end{align*}

\begin{align*}
\text{from read-only-unowned [of } 0 \text{ Suc } j] \ j\text{-bound } j\text{th} \\
\text{have dist-ro: } & O_j \cap \text{read-only } S = \{
\text{by } \text{force}
\end{align*}

\begin{align*}
\text{show } & \text{thesis} \\
\text{proof} & \text{−}
\end{align*}

−

\begin{align*}
\text{case } & \text{True} \\
\text{with dist-a-owns show } & \text{False by } \text{auto}
\end{align*}

\begin{align*}
\text{next} \\
\text{case } & \text{False} \\
\text{hence } a \notin & (O \cup \text{all-acquired } ?\text{take-sb})
\end{align*}

\begin{align*}
\text{using all-acquired-append [of } ?\text{take-sb } ?\text{drop-sb} & \text{]}
\text{by } \text{auto}
\end{align*}

\begin{align*}
\text{from read-only-share-unowned [OF consis-take this a-ro} & \text{]}
\text{have } a \in & \text{read-only } S.
\end{align*}

\begin{align*}
\text{with dist-ro a-owns show } & \text{False by } \text{auto}
\end{align*}

qed

\begin{align*}
\text{qed}
\end{align*}

\begin{align*}
\text{thus } & \text{thesis by } \text{auto}
\end{align*}

qed

qed

\begin{align*}
\text{from Cons.prems} \\
\text{obtain unacq-ts: } & \forall i < \text{length ts. (let } (\ldots, \text{sb}, O, \ldots) = \text{ts!i in }
\begin{align*}
\text{a } \notin & \text{ acquired True (takeWhile (Not } o \text{ is-volatile-Write} sb) O \land }
\text{a } \notin & \text{ all-shared (takeWhile (Not } o \text{ is-volatile-Write} sb) sb) \text{ and }
\text{unacq-sb: } & a \notin \text{ acquired True (takeWhile (Not } o \text{ is-volatile-Write} sb) O \text{ and }
\text{unsh-sb: } & a \notin \text{ all-shared (takeWhile (Not } o \text{ is-volatile-Write} sb) sb}
\end{align*}
\text{apply clarsimp} \\
\text{apply (rule that) } \\
\text{apply (auto simp add: t aargh) } \\
\text{done}
\end{align*}

\begin{align*}
\text{from read-only-unowned [of } 0 \text{] t} \\
\text{have owns-ro: } & O \cap \text{read-only } S = \{
\text{by } \text{force}
\end{align*}

\begin{align*}
\text{from read-only-unacquired-share’ [OF owns-ro consis-take a-ro unsh-sb unacq-sb] & have a } \in \text{ read-only (share (takeWhile (Not } o \text{ is-volatile-Write} sb) sb) } S).
\text{from Cons.hyps [OF dist’ ro-unowned’ consis’ unacq-ts this]} & \text{show } \text{case} \\
\text{by (simp add: t)}
\end{align*}
lemma not-shared-not-acquired-switch:
\( \forall X Y. \ (a \notin \text{all-shared}\ sb \ X; a \notin X; a \notin \text{acquired True}\ sb\ X; a \notin Y) \implies a \notin \text{acquired True}\ sb\ Y \)

proof (induct sb)
case Nil thus \( ? \) case by simp
next
case (Cons x sb)
show \( ? \) case
proof (cases x)
case (Write_{\text{ab}}\ volatile\ a'\ sop\ v\ A\ L\ R\ W)
show \( ? \) thesis
proof (cases volatile)
case True
from Cons.prems obtain
a-X: a \notin X and a-acq: a \notin \text{acquired True}\ sb\ (X \cup A - R) and
a-Y: a \notin Y and a-R: a \notin R and
a-shared: a \notin \text{all-shared}\ sb
by (clarsimp simp add: Write_{\text{ab}}\ True)
show \( ? \) thesis
proof (cases a \in A)
case True
with a-X a-R
have a \in X \cup A - R by auto
from all-shared-acquired-in [OF this \ a-shared] have a \in \text{acquired True}\ sb\ (X \cup A - R),
with a-acq have False by simp
thus \( ? \) thesis ..
next
case False
with a-X a-Y obtain a-X': a \notin X \cup A - R and a-Y': a \notin Y \cup A - R
by auto
from Cons.hyps [OF a-shared a-X' a-acq a-Y'] show \( ? \) thesis
by (auto simp add: Write_{\text{ab}}\ True)
qed
next
case False with Cons.hyps [of X Y] Cons.prems show \( ? \) thesis by (auto simp add: Write_{\text{ab}})
qed
next
case Read_{\text{ab}} with Cons.hyps [of X Y] Cons.prems show \( ? \) thesis by (auto)
next
case Prog_{\text{ab}} with Cons.hyps [of X Y] Cons.prems show \( ? \) thesis by (auto)
next
case (Ghost_{\text{ab}} A L R W)
from Cons.prems obtain
a-X: a \notin X and a-acq: a \notin \text{acquired True}\ sb\ (X \cup A - R) and
a-Y: a \notin Y and a-R: a \notin R and
a-shared: a \notin \text{all-shared}\ sb
by (clarsimp simp add: Ghost_{\text{ab}})
show \( ? \) thesis
proof (cases a \in A)
case True
with a-X a-R
have a \in X \cup A - R by auto
from all-shared-acquired-in [OF this \ a-shared] have a \in \text{acquired True}\ sb\ (X \cup A - R),
with a-acq have False by simp

422
thus ?thesis ..
next
case False
with a-X a-Y obtain a-X': a /∈ X ∪ A − R and a-Y': a /∈ Y ∪ A − R
by auto
from Cons.hyps [OF a-shared a-X' a-acq a-Y']
show ?thesis
  by (auto simp add: Ghosts)
qed
qed

lemmas read-only-share-all-acquired-in ':
\[\forall X S. \left(\{X\} \cap \text{read-only}\ S = \{}\right);\ \text{weak-sharing-consistent}\ \O\ \text{sb};\ a \in \text{read-only}\ (\text{share sb } S)\] 
\[\Rightarrow a \in \text{read-only} (\text{share sb Map.empty}) \lor (a \in \text{read-only} S \land a \notin \text{acquired True sb } \O \land a \notin \text{all-shared sb})\]
proof (induct sb)
case Nil thus ?case by auto
next
case (Cons x sb)
show ?case
proof (cases x)
  case (Write\textsubscript{w} volatile a' sop v A L R W)
  show ?thesis
  proof (cases volatile)
  case True
  note volatile=this
  from Cons.prems
  obtain a-in: a ∈ \text{read-only} (\text{share sb } (S \oplus R \ominus A \cap L)) \land 
  owns-ro: S \cap \text{read-only} S = \{} \land 
  L-A: L ⊆ A \land A-R: A \cap R = \{} \land R-owne : R \subseteq \O \land 
  consi's: weak-sharing-consistent \ (\O \cup A − R) sb 
  by (clarsimp simp add: Write\textsubscript{w} True)
  from owns-ro A-R R-owns have owns-ro': (\O \cup A − R) \cap \text{read-only} (S \oplus R \ominus A \cap L) = \{}
  by (auto simp add: in-read-only-cons)
    from Cons.hyps [OF owns-ro' consi's a-in]
    have hyp: a ∈ \text{read-only} (\text{share sb Map.empty}) \lor 
    \ (a ∈ \text{read-only} (S \oplus R \ominus A \cap L) \land a \notin \text{acquired True sb } (\O \cup A − R) \land a \notin \text{all-shared sb}).
    have a ∈ \text{read-only} (\text{share sb } (\text{Map.empty} \oplus S \ominus R \ominus A \cap L)) \lor 
    \ (a ∈ \text{read-only} S \land a \notin \text{acquired True sb } (\O \cup A − R) \land a \notin \text{all-shared sb})
    proof
    
    \{ 
    assume a-emp: a ∈ \text{read-only} (\text{share sb Map.empty})
    have read-only Map.empty ⊆ \text{read-only} (\text{Map.empty} \oplus R \ominus A \cap L)
      by (auto simp add: in-read-only-cons)
    from share-read-only-mono-in [OF a-emp this]
    have a ∈ \text{read-only} (\text{share sb } (\text{Map.empty} \oplus S \ominus R \ominus A \cap L)).
    \}
  moreover
  
  \{ 
  assume a-ro: a ∈ \text{read-only} (S \oplus R \ominus A \cap L) \land 
  a-not-acq: a \notin \text{acquired True sb } (\O \cup A − R) \land 
  a-unsh: a \notin \text{all-shared sb}
  have ?thesis
  proof (cases a ∈ \text{read-only} S)
  case True
  with a-ro obtain a-A: a \notin A

by (auto simp add: in-read-only-convs)
  with True a-not-acq a-unsh R-owns owns-ro
  show ?thesis
  by auto
next
case False
with a-ro have a-ro-empty: a ∈ read-only (Map.empty ⊕ W R ⊖ A L)
by (auto simp add: in-read-only-convs split: if-split-asm)

have read-only (Map.empty ⊕ W R ⊖ A L) ⊆ read-only (S ⊕ W R ⊖ A L)
by (auto simp add: in-read-only-convs)
with owns-ro'
have owns-ro-empty: (O ∪ A − R) ∩ read-only (Map.empty ⊕ W R ⊖ A L) = {}
by blast

from read-only-unacquired-share' [OF owns-ro-empty consis' a-ro-empty a-not-acq]
have a ∈ read-only (share sb (Map.empty ⊕ W R ⊖ A L)),
  thus ?thesis
  by simp
qed

moreover note hyp
ultimately show ?thesis by blast
qed

then show ?thesis
by (clarsimp simp add: Write_true True)
next
case False with Cons show ?thesis
by (auto simp add: Write_true)

next
case Read_true with Cons show ?thesis by auto
next
case Prog_true with Cons show ?thesis by auto
next
case (Ghost_true A L R W)
from Cons.prems
obtain a-in: a ∈ read-only (share sb (S ⊕ W R ⊖ A L)) and
  owns-ro: O ∩ read-only S = {} and
  L-A: L ⊆ A and A-R: A ∩ R = {} and R owns: R ⊆ O and
  consis': weak-sharing-consistent (O ∪ A − R) sb
by (clarsimp simp add: Ghost_true)

from owns-ro A-R R owns have owns-ro': (O ∪ A − R) ∩ read-only (S ⊕ W R ⊖ A L) = {}
by (auto simp add: in-read-only-convs)

from Cons.hyps [OF owns-ro' consis' a-in]
have hyp: a ∈ read-only (share sb Map.empty) ∨
  (a ∈ read-only (S ⊕ W R ⊖ A L) ∧ a /∈ acquired True sb (O ∪ A − R) ∧ a /∈ all-shared sb).

have a ∈ read-only (share sb (Map.empty ⊕ W R ⊖ A L)) ∨
  (a ∈ read-only S ∧ a /∈ R ∧ a /∈ acquired True sb (O ∪ A − R) ∧ a /∈ all-shared sb)
proof –
{
  assume a-emp: a ∈ read-only (share sb Map.empty)
  have read-only Map.empty ⊆ read-only (Map.empty ⊕ W R ⊖ A L)
  by (auto simp add: in-read-only-convs)
  from share-read-only-mono-in [OF a-emp this]
have \( a \in \text{read-only} \) (share sb (Map.empty \( \oplus \) R \( \ominus \) L)).
}
moreover
{
assume a-ro: \( a \in \text{read-only} \) (S \( \oplus \) R \( \ominus \) L) and
  a-not-acq: \( a \notin \) acquired True sb (O \( \cup \) A \( \setminus \) R) and
  a-unsh: \( a \notin \) all-shared sb
have \(?\text{thesis}\)
proof (cases \( a \in \text{read-only} \) S)
case True
with a-ro obtain a-A: \( a \notin A \) by (auto simp add: in-read-only-convs)
  show \(?\text{thesis}\) by auto
next
case False
with a-ro have a-ro-empty: \( a \in \text{read-only} \) (Map.empty \( \oplus \) R \( \ominus \) L)
by (auto simp add: in-read-only-convs split: if-split-asm)

have \(?\text{thesis}\) by simp qed

moreover note hyp
ultimately show \(?\text{thesis}\) by blast
qed

then show \(?\text{thesis}\) by (clarsimp simp add: Ghost sb)
qed

lemma in-read-only-share-all-until-volatile-write':
assumes dist: ownership-distinct ts
assumes consis: sharing-consis S ts
assumes ro-unowned: read-only-unowned S ts
assumes i-bound: i < length ts
assumes ts-i: \( \text{ts-i} = (p, is, \theta, sb, D, O, R) \)
assumes a-unacquired-others: \( \forall j < \text{length ts}. i \neq j \rightarrow \)
  (let \((\ldots, sb_j, \ldots, O, \cdot) = \text{ts-i}\) in
  a \( \notin \) acquired True (takeWhile (Not \( \circ \) is-volatile-Write sb) sb_j) O \( \wedge \)
  a \( \notin \) all-shared (takeWhile (Not \( \circ \) is-volatile-Write sb) sb_j))
assumes a-ro-share: \( a \in \text{read-only} \) (share sb S)
shows a \( \in \text{read-only} \) (share (dropWhile (Not \( \circ \) is-volatile-Write sb) sb)
  (share-all-until-volatile-write ts S))
proof
  from consis
  interpret sharing-consis S ts.
  interpret read-only-unowned S ts by fact
  from sharing-consis [OF i-bound ts-i]
have consis-sb: sharing-consistent $S \cap \text{sb}$.

from sharing-consistent-weak-sharing-consistent [OF this]
have weak-consis: weak-sharing-consistent $O \cap \text{sb}$.

from read-only-unowned [OF i-bound ts-i]
have owns-ro: $O \cap \text{read-only} = \{\}$.

from read-only-share-all-acquired-in’ [OF owns-ro weak-consis a-ro-share]

have $a \in \text{read-only} (\text{share sb Map.empty}) \lor a \in \text{read-only} S \land a \notin \text{acquired True sb} O \land a \notin \text{all-shared} \text{ sb}.

moreover

let $?\text{take-sb} = (\text{takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) \text{ sb})$
let $?\text{drop-sb} = (\text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb} ) \text{ sb})$

from weak-consis weak-sharing-consistent-append [OF $?\text{take-sb} ?\text{drop-sb} ]$

obtain weak-consis’: weak-sharing-consistent (acquired True $?\text{take-sb} O$) $?\text{drop-sb}$ and
weak-consis-take: weak-sharing-consistent $O ?\text{take-sb}$

by auto

{ 
assume $a \in \text{read-only} (\text{share sb Map.empty})$
with share-append [of $?\text{take-sb} ?\text{drop-sb}$]
have $a$-in’: $a \in \text{read-only} (\text{share } ?\text{drop-sb} (\text{share } ?\text{take-sb Map.empty}))$

by auto

have owns-empty: $O \cap \text{read-only Map.empty} = \{\}$
by auto

from weak-sharing-consistent-preserves-distinct [OF weak-consis-take owns-empty]
have acquired True $?\text{take-sb} O \cap \text{read-only} (\text{share } ?\text{take-sb Map.empty}) = \{\}$.

from read-only-share-all-acquired-in [OF this weak-consis’ a-in’]
have $a \in \text{read-only} (\text{share } ?\text{drop-sb Map.empty}) \lor a \in \text{read-only} (\text{share } ?\text{take-sb Map.empty}) \land a \notin \text{all-acquired } ?\text{drop-sb}$.

moreover

{ 
assume $a$-ro-drop: $a \in \text{read-only} (\text{share } ?\text{drop-sb Map.empty})$

have read-only Map.empty $\subseteq$ read-only (share-all-until-volatile-write ts $S$)
by auto

from share-read-only-mono-in [OF a-ro-drop this]

have $?\text{thesis}$.
}

moreover

{ 
assume $a$-ro-take: $a \in \text{read-only} (\text{share } ?\text{take-sb Map.empty})$
assume a-unacq-drop: $a \notin \text{all-acquired } ?\text{drop-sb}$
from read-only-share-unowned-in [OF weak-consis-take a-ro-take]

have $a \in O \cup \text{all-acquired } ?\text{take-sb}$ by auto

hence $a \in O \cup \text{all-acquired sb}$ using all-acquired-append [of $?\text{take-sb} ?\text{drop-sb}$]
by auto

from share-all-until-volatile-write-thread-local’ [OF dist consis i-bound ts-i this] a-ro-share

have $?\text{thesis}$ by (auto simp add: read-only-def)
}

ultimately have $?\text{thesis}$ by blast

}

moreover

{ 
assume a-ro: $a \in \text{read-only } S$
assume a-unacq: $a \notin \text{acquired True sb } O$
assume a-unsh: $a \notin \text{all-shared sb}$

}
with all-shared-append [of ?take-sb ?drop-sb]
obtain a-notin-append: a /∈ all-shared ?take-sb ?drop-sb
by auto
have ?thesis
proof (cases a ∈ acquired True ?take-sb ?drop-sb O)
case True
from all-shared-acquired-in [OF this a-notin-drop] acquired-append [of True ?take-sb ?drop-sb O] a-unacq
have False
by auto
thus ?thesis ..
ext
next case False
with a-unacquired-others i-bound ts-i a-notin-take
have a-unacq': ∀ j < length ts.
  (let (i, sbj, O) = ts! j in
   a /∈ acquired True (takeWhile (Not ◦ is-volatile-Write sbj) sbj) O ∧
   a /∈ all-shared (takeWhile (Not ◦ is-volatile-Write sbj) sbj)
  )
by (auto simp add: Let-def)

from local.weak-sharing-consis-axioms have weak-sharing-consis ts .
from read-only-share-all-until-volatile-write-unacquired' [OF dist ro-unowned
  weak-sharing-consis ts: a-unacq' a-ro]
have a-ro-all: a ∈ read-only (share-all-until-volatile-write ts S) .

from weak-consis weak-sharing-consistent-append [of O ?take-sb ?drop-sb]
have weak-consis-drop: weak-sharing-consistent (acquired True ?take-sb ?drop-sb O) ?drop-sb
by auto

from weak-sharing-consistent-preserves-distinct-share-all-until-volatile-write [OF dist
  ro-unowned (weak-sharing-consis ts: i-bound ts-i]
have acquired True ?take-sb O ∩
  read-only (share-all-until-volatile-write ts S) = {}. 

from read-only-unacquired-share' [OF this weak-consis-drop a-ro-all a-notin-drop]
  acquired-append [of True ?take-sb ?drop-sb O] a-unacq
show ?thesis by auto
qed
}
ultimately show ?thesis by blast
qed

lemma all-acquired-unshared-acquired:
  ∀ O. a ∈ all-acquired sb ==⇒ a /∈ all-shared sb ==⇒ a ∈ acquired True sb O
apply (induct sb)
apply (auto split: memref.split intro: all-shared-acquired-in)
done

lemma safe-RMW-common:
  assumes safe: Os Rs i-→ (RMW a t (D,f) cond ret A L R W # is, θ, m, D, O, S) √
shows (a ∈ O ∨ a ∈ dom S) ∧ (∀ j < length Os. i≠j −→ (Rs!j) a ≠ Some False)
using safe
apply (cases)
apply (auto simp add: domIff)
done

lemma acquired-reads-all-acquired': ∀ O.
  acquired-reads True sb O ⊆ acquired True sb O ∪ all-shared sb
apply (induct sb)
apply clarsimp
apply (auto split: memref.splits dest: all-shared-acquired-in)
done

lemma release-all-shared-exchange:
\[ \forall S' R. \forall a \in \text{all-shared sb.} (a \in S') \implies \text{release sb } S' R = \text{release sb } S R \]
proof (induct sb)
case Nil thus \(?\) case by auto
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write sb volatile a' sop v A L R W)
    show ?thesis
    proof
      with Cons.
      show ?thesis
      by (auto simp add: Write sb volatile)
    qed
  qed
next
  case Read sb with Cons show ?thesis by auto
next
  case Prog sb with Cons show ?thesis by auto
next
  case (Ghost sb A L R W)
  from augment-rels-shared-exchange \[\text{of } R S S' R'\]
  have augment-rels \[S' R R' = \text{augment-rels } S R R'\]
  by (auto simp add: Ghost sb)
  with Cons.
  show ?thesis
  by (auto simp add: Ghost sb)
qed

lemma release-append-Prog:
\[ \forall S R. \text{(release } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write sb}) (sb @ [\text{Prog sb } p_1 p_2 \text{ mis}])) S R) = \]
\[ (\text{release } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write sb}) sb) S R) \]
by (induct sb) (auto split: memref.splits)

A.5 Simulation of Store Buffer Machine with History by Virtual Machine with Delayed Releases

theorem (in xvalid-program) concurrent-direct-steps-simulates-store-buffer-history-step:
assumes step-sb: \((ts_{sb},m_{sb},S_{sb}) \Rightarrow_{sb} (ts'_{sb},m'_{sb},S'_{sb})\)
assumes valid-own: valid-ownership \[S_{sb} t_{sb}\]
assumes valid-sb-reads: valid-reads \[m_{sb} t_{sb}\]
assumes valid-hist: valid-history program-step \[t_{sb}\]
assumes valid-sharing: valid-sharing \[S_{sb} t_{sb}\]
assumes tmps-distinct: tmps-distinct \[t_{sb}\]
assumes valid-sops: valid-sops \[t_{sb}\]
assumes valid-dd: valid-data-dependency \[t_{sb}\]
assumes load-tmps-fresh: load-tmps-fresh ts_{sb}
assumes enough-flushs: enough-flushs ts_{sb}
assumes valid-program-history: valid-program-history ts_{sb}
assumes valid: valid ts_{sb}
assumes sim: (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)
assumes safe-reach: safe-reach-direct safe-delayed (ts, m, S)
shows valid-ownership S_{sb} \land ts_{sb} \land valid-reads m_{sb} \land ts_{sb} \land valid-history program-step ts_{sb} \land
valid-sharing S_{sb} \land tmpps-distinct ts_{sb} \land valid-data-dependency ts_{sb} \land
valid-sops ts_{sb} \land load-tmps-fresh ts_{sb} \land enough-flushs ts_{sb} \land
valid-program-history ts_{sb} \land valid ts_{sb} \land
(\exists ts', S', m'. (ts, m, S) \Rightarrow^* (ts', m', S') \land
(ts', m', S') \sim (ts', m', S'))

proof –

interpret direct-computation:
  computation direct-memop-step empty-storebuffer-step program-step \lambda p p' is sb. sb .
interpret sbh-computation:
  computation sbh-memop-step flush-step program-step
  \lambda p p' is sb @ [Prog sb p p' is] .
interpret valid-ownership S_{sb} ts_{sb} by fact
interpret valid-reads m_{sb} ts_{sb} by fact
interpret valid-history program-step ts_{sb} by fact
interpret valid-sharing S_{sb} ts_{sb} by fact
interpret tmpps-distinct ts_{sb} by fact
interpret valid-sops ts_{sb} by fact
interpret valid-data-dependency ts_{sb} by fact
interpret load-tmps-fresh ts_{sb} by fact
interpret enough-flushs ts_{sb} by fact
interpret valid-program-history ts_{sb} by fact
from valid-own valid-sharing
have valid-own-sharing: valid-ownership-and-sharing S_{sb} ts_{sb}
  by (simp add: valid-sharing-def valid-ownership-and-sharing-def)
then
interpret valid-ownership-and-sharing S_{sb} ts_{sb} .

from safe-reach-safe-refl [OF safe-reach]
have safe: safe-delayed (ts, m, S).

from step-sb
show \theta thesis
proof (cases)
case (Memop i p_{sb} is_{sb} \partial_{sb} sb \ D_{sb} O_{sb} R_{sb} \ is_{sb}' \partial_{sb}' sb' \ D_{sb}' O_{sb}' R_{sb}')
then obtain
ts_{sb}': ts_{sb} = ts_{sb}[i := (p_{sb}, is_{sb}', \partial_{sb}', sb', D_{sb}', O_{sb}', R_{sb}')] and
i-bound: i < length ts_{sb} and
ts_{sb}': i = (p_{sb}, is_{sb}', \partial_{sb}', sb', D_{sb}', O_{sb}', R_{sb}') and
sbh-step: (is_{sb}', \partial_{sb}', sb', m_{sb}', D_{sb}', O_{sb}', R_{sb}', S_{sb}) \rightarrow sbh
  (is_{sb}', \partial_{sb}', sb', m_{sb}', D_{sb}', O_{sb}', R_{sb}', S_{sb})

429
by auto

from sim obtain
m: m = flush-all-until-volatile-write ts sb m sb and
S: S = share-all-until-volatile-write ts sb S sb and
leq: length ts sb = length ts and

ts-sim: \forall i < \text{length ts sb}.
\text{let (p, is sb, o, D sb, O sb, R) = ts sb ! i;}
suspends = dropWhile (Not o is-volatile-Write sb) sb
\text{in } \exists is D. instrs suspends @ is sb = is @ prog-instrs suspends \land
D sb = (D \lor \text{outstanding-refs is-volatile-Write sb sb } \neq \{\}) \land
ts i =
\text{(hd-prog p suspends,}
is, is
\theta \div (\text{dom } \theta - \text{read-tmps suspends}), ()
D, acquired True (takeWhile (Not o is-volatile-Write sb sb) O sb)
\text{release (takeWhile (Not o is-volatile-Write sb sb) (dom S sb) R})
\text{by cases blast}

from i-bound leq have i-bound' i < \text{length ts}
\text{by auto}

\text{have split-sb: sb = takeWhile (Not o is-volatile-Write sb sb) sb \@} dropWhile (Not o is-volatile-Write sb sb)
\text{(is sb = ?take-sb@?drop-sb)}
\text{by simp}

from ts-sim [rule-format, OF i-bound] ts sb-i obtain suspends is D where
suspends: suspends = dropWhile (Not o is-volatile-Write sb sb) sb and
is-sim: instrs suspends @ is sb = is @ prog-instrs suspends and
D: D sb = (D \lor \text{outstanding-refs is-volatile-Write sb sb } \neq \{\}) \land
\text{ts i: ts i =}
\text{(hd-prog p suspends,}
\theta sb \div (\text{dom } \theta sb - \text{read-tmps suspends}), ()
D, acquired True ?take-sb O sb
\text{release ?take-sb (dom S sb) R sb)
\text{by (auto simp add: Let-def)}

from sbh-step-preserves-valid [OF i-bound ts sb-i sbh-step valid]
\text{have valid' valid ts sb'}
\text{by (simp add: ts sb')}

from D have D sb: D sb = (D \lor \text{outstanding-refs is-volatile-Write sb sb } \neq \{\})
\text{apply -}
\text{apply (case-tac outstanding-refs is-volatile-Write sb sb = \{\})}
\text{apply (fastforce simp add: outstanding-refs-conv dest: set-dropWhileD)}
\text{apply (clarsimp)}
\text{apply (drule outstanding-refs-non-empty-dropWhile)}
\text{apply blast}
let \(?ts' = ts[i := (p_{sb}, is_{sb}, \theta_{sb}, {}), D_{sb}, \text{acquired True } sb O_{sb}, release sb (\text{dom } S_{sb}) R_{sb})]\)

have i-bound-ts': i < length ?ts'
  using i-bound'
  by auto

hence ts'^i: ?ts'^i = (p_{sb}, is_{sb}, \theta_{sb}, {}),
  D_{sb}, \text{acquired True } sb O_{sb}, release sb (\text{dom } S_{sb}) R_{sb})
  by simp

from local.sharing-consis-axioms
have sharing-consis-ts_{sb}: sharing-consis S_{sb} ts_{sb}.
from sharing-consis [OF i-bound ts_{sb}-i]
have sharing-consis-sb: sharing-consistent S_{sb} O_{sb} sb.
from sharing-consistent-weak-sharing-consistent [OF this]
have weak-consis-sb: weak-sharing-consistent O_{sb} sb.
from this weak-sharing-consistent-append [of O_{sb} ?take-sb ?drop-sb]
have weak-consis-drop: weak-sharing-consistent (acquired True ?take-sb O_{sb}) ?drop-sb
  by auto
from local.ownership-distinct-axioms
have ownership-distinct-ts_{sb}: ownership-distinct ts_{sb}.
have steps-flush-sb: (ts, m, S) \Rightarrow d^* (?ts', flush ?drop-sb m, share ?drop-sb S)

proof
  from valid-reads [OF i-bound ts_{sb}-i]
  have reads-consis: reads-consistent False O_{sb} m_{sb} sb.
  from reads-consistent-drop-volatile-writes-no-volatile-reads [OF this]
  have no-vol-read: outstanding.refs is-volatile-Read_{sb} ?drop-sb = {}.
  from valid-program-history [OF i-bound ts_{sb}-i]
  have causal-program-history is_{sb} sb.
  then have cph: causal-program-history is_{sb} ?drop-sb
apply
  apply (rule causal-program-history-suffix [where sb=?take-sb] )
  apply (simp)
done
  from valid-last-prog [OF i-bound ts_{sb}-i]
  have last-prog: last-prog p_{sb} sb = p_{sb}.
  then
  have lp: last-prog p_{sb} ?drop-sb = p_{sb}
apply
  apply (rule last-prog-same-append [where sb=?take-sb])
  apply simp
done

from reads-consistent-flush-all-until-volatile-write [OF valid-own-sharing i-bound ts_{sb}-i reads-consis]
  have reads-consis-m: reads-consistent True (acquired True ?take-sb O_{sb}) m ?drop-sb
  by (simp add: m)
from valid-history [OF i-bound ts_{sb}-i]
have h-consis: history-consistent \( \vartheta_{sb} \) \( (\text{hd-prog } p_{sb} \ (?\text{take-sb} @ ?\text{drop-sb})) \)
by (simp)

have last-prog-hd-prog: last-prog \( (\text{hd-prog } p_{sb} \ sb) \) ?take-sb = \( (\text{hd-prog } p_{sb} ~ ?\text{drop-sb}) \)
proof –
from last-prog-hd-prog-append' [OF h-consis] last-prog
have last-prog \( (\text{hd-prog } p_{sb} ~ ?\text{drop-sb}) \) ?take-sb = \( (\text{hd-prog } p_{sb} ~ ?\text{drop-sb}) \)
by (simp)
moreover
have last-prog \( (\text{hd-prog } p_{sb} \ (?\text{take-sb} @ ?\text{drop-sb})) \) ?take-sb =
last-prog \( (\text{hd-prog } p_{sb} ~ ?\text{drop-sb}) \) ?take-sb
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp)
qed

from valid-write-sops [OF i-bound ts_{sb}-i]
have \( \forall \text{sop} \in \text{write-sops } (?\text{take-sb} @ ?\text{drop-sb}). \text{valid-sop} \) \( \text{sop} \)
by (simp)
then obtain valid-sops-take: \( \forall \text{sop} \in \text{write-sops } ?\text{take-sb}. \text{valid-sop} \) \( \text{sop} \)
and valid-sops-drop: \( \forall \text{sop} \in \text{write-sops } ?\text{drop-sb}. \text{valid-sop} \)
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-read-tmps \( (?\text{take-sb} @ ?\text{drop-sb}) \)
by (simp)
then obtain read-tmps-take-drop: read-tmps \( ?\text{take-sb} \cap \text{read-tmps } ?\text{drop-sb} = \{ \} \)
and distinct-read-tmps-drop: distinct-read-tmps \( ?\text{drop-sb} \)
by (simp only: distinct-read-tmps-append)

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]

have hist-consis': history-consistent \( \vartheta_{sb} \) \( (\text{hd-prog } p_{sb} \ ?\text{drop-sb}) \)
by (simp add: last-prog-hd-prog)

have rel-eq: release \( ?\text{drop-sb} \) \( (\text{dom } S) \) (release \( ?\text{take-sb} \) \( (\text{dom } S_{sb} \ R_{sb}) = \)
release \( \text{sb} \) \( (\text{dom } S_{sb} \ R_{sb}) \)
proof –
from release-append [of \( ?\text{take-sb} ?\text{drop-sb} \)]
have release \( \text{sb} \) \( (\text{dom } S_{sb} \ R_{sb}) = \)
release \( ?\text{drop-sb} \) \( (\text{dom } \text{share } ?\text{take-sb} S_{sb})) \) (release \( ?\text{take-sb} \) \( (\text{dom } S_{sb} \ R_{sb}) \)
by simp
also
have dist: ownership-distinct ts_{sb} by fact
have consis: sharing-consis \( S_{sb} \) ts_{sb} by fact
have release ?drop-sb (dom (share ?take-sb $S_{sb}$)) (release ?take-sb (dom $S_{sb}$) $R_{sb}$)

= release ?drop-sb (dom $S$) (release ?take-sb (dom $S_{sb}$) $R_{sb}$)

apply (simp only: $S$)
apply (rule release-shared-exchange-weak [rule-format, OF - weak-consis-drop])
apply (rule share-all-until-volatile-write-thread-local [OF dist consis i-bound ts_{sb}-i, symmetric])

using acquired-all-acquired [of True ?take-sb $O_{sb}$] all-acquired-append [of ?take-sb ?drop-sb]

by auto
finally

show ?thesis by simp

qed

from flush-store-buffer [OF i-bound' is-sim [simplified suspends]
cph ts-i [simplified suspends] refl lp reads-consis-m hist-consis'
valid-sops-drop distinct-read-tmps-drop no-volatile-Read_{sb}-volatile-reads-consistent [OF no-vol-read], of $S$

show ?thesis by (simp add: acquired-take-drop [where pending-write=True, simplified] $D_{sb}$ rel-eq)

qed

from safe-reach-safe-rtrancl [OF safe-reach steps-flush-sb]

have safe-ts': safe-delayed (?ts', flush ?drop-sb m, share ?drop-sb $S$).
from safe-delayedE [OF safe-ts' i-bound-ts' ts'\i]

have safe-memop-flush-sb: map owned ?ts', map released ?ts', i- (is_{sb}, $\theta_{sb}$, flush ?drop-sb m, $D_{sb}$, acquired True sb $O_{sb}$, share ?drop-sb $S$) $\sqrt{\cdot}$.

from acquired-takeWhile-non-volatile-Write_{sb}

have acquired-take-sb: acquired True ?take-sb $O_{sb}$ $\subseteq$ $O_{sb}$ $\cup$ all-acquired ?take-sb .

from sbh-step

show ?thesis

proof (cases)

case (SBHRReadBuffered a v volatile t)

then obtain

is_{sb}: is_{sb} = Read volatile a t $\neq$ is_{sb}' and
$O_{sb}$': $O_{sb}'$ $=$ $O_{sb}$ and
$D_{sb}$': $D_{sb}'$ $=$ $D_{sb}$ and
$\theta_{sb}$': $\theta_{sb}'$ $=$ $\theta_{sb}(t\mapsto v)$ and
sb': sb' =$=$ sb@[Read_{sb} volatile a t v] and
m_{sb}'$: m_{sb}' = m_{sb} and
$S_{sb}$': $S_{sb}'$ $=$ $S_{sb}$ and
$R_{sb}$': $R_{sb}'$ $=$ $R_{sb}$ and
buf-v: buffered-val sb a = Some v

by auto

433
from safe-memop-flush-sb [simplified is sb]

obtain access-cond': a ∈ acquired True sb O sb ∨
a ∈ read-only (share ?drop-sb S) ∨
(volatile ∧ a ∈ dom (share ?drop-sb S)) and
volatile-clean: volatile → ¬ D sb and
rels-cond: ∀ j < length ts. i ≠ j → released (ts!j) a ≠ Some False and
rels-nv-cond: ¬ volatile → (∀ j < length ts. i ≠ j → a /∈ dom (released (ts!j)))
by cases auto

from clean-no-outstanding-volatile-Write sb [OF i-bound ts sb-i] volatile-clean
have volatile-cond: volatile → outstanding-refs is-volatile-Write sb sb = { }
by auto

from buffered-val-witness [OF buf-v] obtain volatile' sop' A' L' R' W'
where
witness: Write sb volatile' a sop' v A' L' R' W' ∈ set sb
by auto

{ 
fix j p j is sbj O j R j D sbj θ sbj sbj
assume j-bound: j < length ts sb
assume neq-i-j: i ≠ j
assume jth: ts sb!j = (p j, is sbj, θ sbj, sbj, D sbj, O j, R j)
assume non-vol: ¬ volatile
have a /∈ O j ∪ all-acquired sb j
proof
assume a-j: a ∈ O j ∪ all-acquired sb j
let ?take-sb j = (takeWhile (Not ◦ is-volatile-Write sb) sb j)
let ?drop-sb j = (dropWhile (Not ◦ is-volatile-Write sb) sb j)

from ts-sim [rule-format, OF j-bound] jth
obtain suspends j is j D j where
suspends j: suspends j = ?drop-sb j and
is j: instrs suspends j @ is sbj = is j @ prog-instrs suspends j and
D j: D sbj = (D j ∨ outstanding-refs is-volatile-Write sb sb j ≠ { }) and
ts j: ts j!j = (hd-prog p j suspends j, is j, θ sbj | (dom θ sbj − read-tmps suspends j), ( ), D j, acquired True ?take-sb j O j, release ?take-sb j (dom S sb) R j)
by (auto simp add: Let-def)

from a-j ownership-distinct [OF i-bound j-bound neq-i-j ts sb-i jth]
have a-notin-sb: a /∈ O sb ∪ all-acquired sb j
by auto
with acquired-all-acquired [of True sb O sb]
have a-not-acq: a ∈ acquired True sb ⋃ sb by blast
with access-cond’ non-vol
have a-ro: a ∈ read-only (share ?drop-sb S)
  by auto
  from read-only-share-unowned-in [OF weak-consis-drop a-ro] a-notin-sb
  acquired-all-acquired [of True ?take-sb ⋃ sb]
  all-acquired-append [of ?take-sb ?drop-sb]
have a-ro-shared: a ∈ read-only S
  by auto
  from rels-nv-cond [rule-format, OF non-vol j-bound [simplified leq] neq-i-j] ts_j
have a ∉ dom (release ?take-sbj (dom (S_sb)) R_j)
  by auto
with dom-release-takeWhile [of sb (dom (S_sb)) R_j]
obtain
  a-rels_j: a ∉ dom R_j and
  a-shared_j: a ∉ all-shared ?take-sbj
  by auto

have a ∉ \bigcup \left( \{ \lambda (-, -, -, sb, -, -). all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) \} \right)
  \set ts_{sb}
proof –
  { 
    fix k p_k is_k d_k sb_k D_k O_k R_k
    assume k-bound: k < length ts_{sb}
    assume ts-k: ts_{sb} \mid k = (p_k, is_k, d_k, sb_k, D_k, O_k, R_k)
    assume a-in: a ∈ all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_k)
    have False
    proof (cases k=j)
      case True with a-shared_j jth ts-k a-in show False by auto
    next
      case False
      from ownership-distinct [OF j-bound k-bound False [symmetric] jth ts-k] a-j
      have a ∉ (O_k ∪ all-acquired sb_k) by auto
      with all-shared-acquired-or-owned [OF sharing-consis [OF k-bound ts-k]] a-in
      show False
      using all-acquired-append [of takeWhile (Not ◦ is-volatile-Write_{sb}) sb_k]
      dropWhile (Not ◦ is-volatile-Write_{sb}) sb_k]
      all-shared-append [of takeWhile (Not ◦ is-volatile-Write_{sb}) sb_k]
      dropWhile (Not ◦ is-volatile-Write_{sb}) sb_k by auto
    qed
  }
thus ?thesis by (fastforce simp add: in-set-conv-nth)
qed
with a-ro-shared
read-only-shared-all-until-volatile-write-subset’ [of ts_{sb} S_{sb}]
have a-ro-shared_{sb}: a ∈ read-only \exists sb
  by (auto simp add: S)
with read-only-unowned \([OF \ j\text{-bound } j]\)

have a-notin-owns-j: \(a \notin O_j\)

by auto

have own-dist: ownership-distinct \(t_{sb}^{j}\) by fact

have share-consis: sharing-consis \(S_{sb}^{j}\) \(t_{sb}^{j}\) by fact

from sharing-consistent-share-all-until-volatile-write \([OF \ own-dist \ share-consis \ i\text{-bound} \ t_{sb}^{j-1}]\)

have consis': sharing-consistent \(S\) (acquired True \(?\text{take-sb}_{sb}\) \(?\text{drop-sb}\)

by (simp add: \(S\))

from share-all-until-volatile-write-thread-local \([OF \ own-dist \ share-consis \ j\text{-bound} \ jh \ a\text{-j}]\) a-ro-shared

have a-ro-take: \(a \in \text{read-only} \ (\text{share} \ ?\text{take-sb}_{j} \ S_{sb})\)

by (auto simp add: domIff \(S\) read-only-def)

from sharing-consis \([OF \ j\text{-bound } jh]\)

have sharing-consistent \(S_{sb}^{j}\) \(O_j^{sb}\) \(sb^j\).

from sharing-consistent-weak-sharing-consistent \([OF \ this]\)

weak-sharing-consistent-append \([of \ O_j \ ?\text{take-sb}_{j} \ ?\text{drop-sb}_{j}]\)

have weak-consis-drop:weak-sharing-consistent \(O_j^{sb}\) \(?\text{take-sb}_{j}\)

by auto

from read-only-share-acquired-all-shared \([OF \ this \ \text{read-only-unowned} \ [OF \ j\text{-bound} \ jh] \ a\text{-ro-take} \ ] \ a\text{-notin-owns-j} \ a\text{-shared}_{j}\)

have a \(\notin \ \text{all-acquired} \ ?\text{take-sb}_{j}\)

by auto

with a-j a-notin-owns-j

have a-drop: \(a \in \text{all-acquired} \ ?\text{drop-sb}_{j}\)

using all-acquired-append \([of \ ?\text{take-sb}_{j} \ ?\text{drop-sb}_{j}]\)

by simp

from i-bound j-bound leq have j-bound-ts': \(j < \text{length} \ ?ts'\)

by auto

note conflict-drop = a-drop \([\text{simplified} \ \text{suspends}_j \ \text{symmetric}]\)

from split-all-acquired-in \([OF \ \text{conflict-drop}]\)

show False

proof

assume \(\exists \text{sop} \ a' \ v \ ys \ zs \ A \ L \ R \ W\).

(suspends_j = ys @ Write_{sb} \ True \ a' \ sop \ v \ A \ L \ R \ W\# \ zs) \land a \in A

then obtain \(a' \ \text{sop}' \ v' \ ys \ zs \ A' \ L' \ R' \ W'\) where

split-suspends_j: suspends_j = ys @ Write_{sb} \ True \ a' \ sop' \ v' \ A' \ L' \ R' \ W'\# \ zs

(is suspends_j = ?suspends) and

a-A': \(a \in A'\)

by blast

from sharing-consis \([OF \ j\text{-bound } jh]\)

436
have sharing-consistent $S_{sb} \circ_j sb_j$.
then have $A' \cap R'$ $\cap \{\}$
by (simp add: sharing-consistent-append [of - $?take-sb_j \circ_drop-sb_j$, simplified]
suspends$|$ symmetric$|$ split-suspends$|$ sharing-consistent-append)

from valid-program-history [OF j-bound jth]
have causal-program-history is$sb_j sb_j$. 
then have cph: causal-program-history is$sb_j sb_j$.

apply 
apply (rule causal-program-history-suffix [where sb=?take-sb]
apply (simp only: split-suspends$|$ symmetric$|$ suspends)
apply (simp add: split-suspends)
done

from ts$neq_i-j j-bound$
have ts$'j$ : ts$'j$ = (hd prog $p_j suspends_j$, is$,$
$v_{sb} \setminus (dom v_{sb} \circ_read-tmps suspends_j),(),$
$D_j$, acquired True $?take-sb_j O_j$, release $?take-sb_j \circ_dom S_{sb} R_j$)
by auto
from valid-last-prog [OF j-bound jth] have last-prog: last-prog $p_j sb_j = p_j$.
then have lp: last-prog $p_j suspends_j = p_j$
apply 
apply (rule last-prog-same-append [where sb=?take-sb]
apply (simp only: split-suspends$|$ symmetric$|$ suspends)
apply simp
done
from valid-reads [OF j-bound jth]
have reads-consis-j: reads-consistent False $O_j m_{sb} sb_j$.
from reads-consistent-flush-all-until-volatile-write [OF \valid-ownership-and-sharing
$S_{sb} ts_{sb} \circ_j j-bound$
$\circ_j reads-consis-j$]
have reads-consis-m-j: reads-consistent True (acquired True $?take-sb_j O_j$) $m suspends_j$
by (simp add: $m suspends_j$)

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound neq-i-j ts$sb_i jth$
have outstanding-ref is-Write$_{sb} \circ_drop-sb \cap outstanding-ref is-non-volatile-Read$_{sb}$
suspends$ = \{\}$
by (simp add: sus$ppends_j$)

from reads-consistent-flush-independent [OF this reads-consis-m-j]
have reads-consis-flush-suspend: reads-consistent True (acquired True $?take-sb_j O_j$)
(flush $?drop-sb m$) suspends$.$
hence reads-consis-ys: reads-consistent True (acquired True $?take-sb_j O_j$)
(flush $?drop-sb m$) (ys@[Write$_{sb} True a' sop' v' A' L' R' W'])
by (simp add: split-suspends$_j$ reads-consistent-append)

from valid-write-sops [OF j-bound jth]
have $\forall sop \in write-sops (\?take-sb_j \circ _\circ ?suspects). valid-sop sop$
by (simp add: split-suspends$_j$ [symmetric] suspends$_j$)
then obtain \( \forall \) sop\( \in \) write-sops ?take-sb\( j \), valid-sop sop and 
valid-sops-drop: \( \forall \) sop\( \in \) write-sops (ys@\([\text{Write}_{ab} \ True \ a' \ sop' \ v' \ A' L' R' W']\)). valid-sop sop 
apply (simp only: write-sops-append) 
apply auto 
doneda

from read-tmps-distinct [OF j-bound jth] 
have distinct-read-tmps (?take-sb\( j \)@suspends\( j \)) 
  by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
then obtain 
read-tmps-take-drop: read-tmps ?take-sb\( j \) \( \cap \) read-tmps suspends\( j \) = \( \{} \) and 
distinct-read-tmps-drop: distinct-read-tmps suspends\( j \) 
apply (simp only: split-suspends\( j \) [symmetric] suspends\( j \)) 
apply (simp only: distinct-read-tmps-append) 
doneda

from valid-history [OF j-bound jth] 
have h-consis: 
  history-consistent \( \theta_{abj} \) (hd-prog \( p_j \) (?take-sb\( j \)@suspends\( j \))) (?take-sb\( j \)@suspends\( j \)) 
apPLY (simp only: split-suspends\( j \) [symmetric] suspends\( j \)) 
apPLY simp 
doneda

have last-prog-hd-prog: last-prog (hd-prog \( p_j \) sb\( j \)) ?take-sb\( j \) = (hd-prog \( p_j \) suspends\( j \)) 
proof – 
from last-prog have last-prog \( p_j \) (?take-sb\( j \)@?drop-sb\( j \)) = \( p_j \) 
by simp 
from last-prog-hd-prog-append' [OF h-consis] this 
have last-prog (hd-prog \( p_j \) suspends\( j \)) ?take-sb\( j \) = hd-prog \( p_j \) suspends\( j \) 
by (simp only: split-suspends\( j \) [symmetric] suspends\( j \)) 
mOREOVER 
have last-prog (hd-prog \( p_j \) (?take-sb\( j \)@suspends\( j \))) ?take-sb\( j \) = last-prog (hd-prog \( p_j \) suspends\( j \)) ?take-sb\( j \) 
apPLY (simp only: split-suspends\( j \) [symmetric] suspends\( j \)) 
by (rule last-prog-hd-prog-append) 
ultimately show ?thesis 
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \)) 
qeda

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop 
h-consis] last-prog-hd-prog 
have hist-consis : history-consistent \( \theta_{abj} \) (hd-prog \( p_j \) suspends\( j \)) suspends\( j \) 
  by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
from reads-consistent-drop-volatile-writes-no-volatile-reads 
[OF reads-consis-j] 
have no-vol-read: outstanding-refs is-volatile-Read\( ab \) 
  (ys@\([\text{Write}_{ab} \ True \ a' \ sop' \ v' \ A' L' R' W']\)) = \( \{} \) 
  by (auto simp add: outstanding-refs-append suspends\( j \) [symmetric] suspends\( j \))

438
have acq-simp:
acquired True (ys @ [Write sb True a' sop' v' A' L' R' W'])
  (acquired True ?take-sb j Oj) =
  acquired True ys (acquired True ?take-sb j Oj) \cup A' \rightarrow R'
by (simp add: acquired-append)

from flush-store-buffer-append [where sb=ys@] [Write sb True a' sop' v' A' L' R' W']
and sb'=zs, simplified,
  OF j-bound-ts' isj [simplified split-suspends] cph [simplified suspends]
ts'j [simplified split-suspends] refl lp [simplified split-suspends] reads-consis-ys
  hist-consis' [simplified split-suspends] valid-sops-drop
  distinct-read-tmps-drop [simplified split-suspends]
  no-volatile-Read sb-volatile-reads-consistent [OF no-vol-read], where
  S=share ?drop-sb S

obtain isj' \mathcal{R}_j' where
  isj'': instrs zs @ isj' = isj' @ prog-instrs zs and
  steps-ys: (?ts', flush ?drop-sb m, share ?drop-sb S) \Rightarrow d^*
  (?ts'j'':=(last-prog
    (hd-prog p j (Write sb True a' sop' v' A' L' R' W'\# zs)) (ys@[Write sb True a'
    sop' v' A' L' R' W']))),
  isj',
  \theta_{sbj} \upharpoonright dom \theta_{sbj} = \text{read-tmps zs},
  (l, True, acquired True ys (acquired True ?take-sb j Oj) \cup A' \rightarrow R', \mathcal{R}_j'),
  flush (ys@[Write sb True a' sop' v' A' L' R' W']) (flush ?drop-sb m),
  share (ys@[Write sb True a' sop' v' A' L' R' W']) (share ?drop-sb S))
  (is (-,-,-) \Rightarrow d^* (?ts-ys,?m-ys,?shared-ys))

by (auto simp add: acquired-append outstanding-refs-append)

from i-bound'i have i-bound-ys: i < length ?ts-ys
by auto

from i-bound'i neq-i-j
have ts-ys-i: (?ts-ys!i = (p sb, is sb, \theta sb,()),
  \mathcal{D}_{sb}, acquired True sb O sb, release sb (dom S sb) \mathcal{R}_{sb})
by simp

note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is sb]
non-vol a-not-acq
have a \in read-only (share (ys@[Write sb True a' sop' v' A' L' R' W']) (share ?drop-sb S))
apply cases
apply (auto simp add: Let-def is sb)
done

439
with a-A'
show False by (simp add: share-append in-read-only-convs)

next
assume ∃ A L R W ys zs. suspends j = ys @ Ghost sb j A L R W # zs ∧ a ∈ A
then obtain A' L' R' W' ys zs where
  split-suspends j: suspends j = ys @ Ghost sb j A' L' R' W' # zs
  (is suspends j = ?suspends) and
a-A': a ∈ A'
by blast

from valid-program-history [OF j-bound jth]
have causal-program-history is sb j.
then have cph: causal-program-history is sb j ?suspends
  apply –
  apply (rule causal-program-history-suffix [where sb = ?take-sb j ] )
  apply (simp only: split-suspends j [symmetric] suspends j)
  apply (simp add: split-suspends j)
done

from ts j neq-i-j j-bound
have ts' j: ?ts' j = (hd prog p j suspends j, is j,
  ?sb j | (dom ?sb j − read-tmps suspends j),).
   D j, acquired True ?take-sb j O j, release ?take-sb j (dom S sb j) R j
   by auto
from valid-last-prog [OF j-bound jth] have last-prog: last-prog p j sb j = p j.
then
have lp: last-prog p j suspends j = p j
  apply –
  apply (rule last-prog-same-append [where sb = ?take-sb j ])
  apply (simp only: split-suspends j [symmetric] suspends j)
  apply simp
done

from valid-reads [OF j-bound jth]
have reads-consis-j: reads-consistent False O j m sb j.
   from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing
   S sb j ts sb j ] j-bound
   jth reads-consis-j]
   have reads-consis-m-j: reads-consistent True (acquired True ?take-sb j O j) m suspends j
   by (simp add: m suspends j)

   from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound neq-i-j ts sb j ]
   jth]
   have outstanding-reads is-Write sb ?drop-sb ∩ outstanding-reads is-non-volatile-Read sb
   suspends j = {}
   by (simp add: suspends j)
   from reads-consistent-flush-independent [OF this reads-consis-m-j]
have \( \text{reads-consis-flush-suspend}: \) reads-consistent True (acquired True ?take-sbj \( C_j \))
(\text{flush } ?\text{drop-sb m}) \text{suspends}_j.

hence \( \text{reads-consis-ys}: \) reads-consistent True (acquired True ?take-sbj \( C_j \))
(\text{flush } ?\text{drop-sb m}) \text{ys}@[\text{Ghost}_{sb} A' L' R' W'])
by (simp add: split-suspends \( j \) \text{reads-consistent-append})
from \text{valid-write-sops} [OF \( j \text{-bound } j \)]
have \( \forall \text{sop}\in\text{write-sops} (?\text{take-sb}_j ?\text{?suspends}_j) \). \text{valid-sop} \text{sop}
by (simp add: split-suspends \( j \) \text{symmetric} \text{suspends}_j)
then obtain \text{valid-sops-take}: \( \forall \text{sop}\in\text{write-sops} ?\text{take-sb}_j \). \text{valid-sop} \text{sop}
and \text{valid-sops-drop}: \( \forall \text{sop}\in\text{write-sops} (\text{ys}@[\text{Ghost}_{sb} A' L' R' W']) \). \text{valid-sop} \text{sop}
apply (simp only: \text{write-sops-append})
apply auto
done

from \text{read-tmps-distinct} [OF \( j \text{-bound } j \)]
have \( \text{distinct-read-tmps} (?\text{take-sb}_j ?\text{suspends}_j) \)
by (simp add: split-suspends \( j \) \text{symmetric} \text{suspends}_j)
then obtain \text{read-tmps-take-drop}: \( \text{read-tmps} ?\text{take-sb}_j \cap \text{read-tmps} \text{suspends}_j = \{\} \) and
\text{distinct-read-tmps-drop}: \( \text{distinct-read-tmps} \text{suspends}_j \)
apply (simp only: \text{split-suspends \( j \) symmetric} \text{suspends}_j)
apply (simp only: \text{distinct-read-tmps-append})
done

from \text{valid-history} [OF \( j \text{-bound } j \)]
have \( \text{h-consis}: \)
\text{history-consistent} \( \theta_{sbj} (\text{hd-prog} p_j (?\text{take-sb}_j ?\text{suspends}_j)) (?\text{take-sb}_j ?\text{suspends}_j) \)
apply (simp only: \text{split-suspends \( j \) symmetric} \text{suspends}_j)
apply simp
done

have \( \text{last-prog-hd-prog}: \) \text{last-prog} (\text{hd-prog} p_j sbj) ?\text{take-sb}_j = (\text{hd-prog} p_j \text{suspends}_j)
proof –
from \text{last-prog} have \( \text{last-prog} p_j (?\text{take-sb}_j ?\text{?drop-sb}_j) = p_j \)
by simp
from \text{last-prog-hd-prog-append' }[\text{OF \text{h-consis}}] \) this
have \( \text{last-prog} (\text{hd-prog} p_j \text{suspends}_j) ?\text{take-sb}_j = \text{hd-prog} p_j \text{suspends}_j \)
by (simp only: \text{split-suspends \( j \) symmetric} \text{suspends}_j)
moreover
have \( \text{last-prog} (\text{hd-prog} p_j (?\text{take-sb}_j @ \text{suspends}_j)) ?\text{take-sb}_j = \text{last-prog} (\text{hd-prog} p_j \text{suspends}_j) ?\text{take-sb}_j \)
apply (simp only: \text{split-suspends \( j \) symmetric} \text{suspends}_j)
by (rule \text{last-prog-hd-prog-append})
ultimately show \( ?\text{thesis} \)
by (simp add: \text{split-suspends \( j \) symmetric} \text{suspends}_j)
qed

from \text{history-consistent-appendD} [OF \text{valid-sops-take read-tmps-take-drop} \text{h-consis}] \text{last-prog-hd-prog}
have hist-consis': history-consistent \( \vartheta_{sbj} \) (hd-prog \( pj \) suspends\( j \)) suspends\( j \)
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))

from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-j]
have no-vol-read: outstanding-refs is-volatile-\( R_{sb} \)
(ys@[Ghost\( sb \) A' L' R' W']) = {}
by (auto simp add: outstanding-refs-append suspends\( j \) [symmetric] split-suspends\( j \))

have acq-simp:
acquired True (ys @[Ghost\( sb \) A' L' R' W'])
(acquired True \( ?take-sb \ O_j \)) =
acquired True ys (acquired True \( ?take-sb \ O_j \) \( \cup \) A' - R')
by (simp add: acquired-append)

from flush-store-buffer-append [where \( sb=ys@[Ghost\( sb \) A' L' R' W'] \) and \( sb'=zs \), simplified,
OF j-bound-ts' \( is_j \) [simplified split-suspends\( j \)] cph [simplified suspends\( j \)]
\( ts' \) [simplified split-suspends\( j \)] refl lp [simplified split-suspends\( j \)] reads-consis-ys
hist-consis' [simplified split-suspends\( j \)] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends\( j \)]
no-volatile-\( \text{Read}_{sb} \)-volatile-reads-consistent [OF no-vol-read, where
\( S=share \ ?drop-sb S \)]

obtain \( is_j' \ R_{j'} \) /where
\( is_j' \) : instrs zs @ \( is_{sbj} \) = \( is_j' \) @ prog-instrs zs and
steps-ys: \( (?ts', flush \ ?drop-sb m, share \ ?drop-sb S) \Rightarrow d^* \)
(\( ?ts'\):j:=last-prog
(\( \text{hd-prog} \ p_j \ (\text{Ghost}_{sb} A' L' R' W' \# zs) \)) (ys@[Ghost\( sb \) A' L' R' W']),
\( is_j' \),
\( \vartheta_{sbj}^{j'} \) (dom \( \vartheta_{sbj} \) - read-tmps zs),
(\).
\( D_j \lor \) outstanding-refs is-volatile-\( \text{Write}_{sb} \) (ys@[Ghost\( sb \) A' L' R' W']) \( \neq \) {},
aquired True ys (acquired True \( ?take-sb \ O_j \) \( \cup \) A' - R',\( R_{j}' \)),
flush (ys@[Ghost\( sb \) A' L' R' W']) (flush \( ?drop-sb m \)),
share (ys@[Ghost\( sb \) A' L' R' W']) (share \( ?drop-sb S \))
(is (-,-,-) \( \Rightarrow d^* \) (?ts-ys,?m-ys,?shared-ys))
by (auto simp add: acquired-append)

from i-bound' have i-bound-ys: i < length ?ts-ys
by auto

from i-bound' neq-i-j
have ts-ys-i: ?ts-ys!i = (\( p_{sb} \), \( is_{sb}, \vartheta_{sb}, () \)),
\( D_{sb} \), acquired True sb \( O_{sb} \), release sb (dom \( S_{sb} \) \( R_{sb} \))
by simp
note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).
from safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is_{sb}] non-vol a-not-acq
have a ∈ read-only (share (ys@[Ghost_{sb} A' L' R' W']) (share ?drop-sb S))
  apply cases
  apply (auto simp add: Let-def is_{sb})
done

with a-A'
show False
  by (simp add: share-append in-read-only-convs)
qed
qed

note non-volatile-unowned-others = this

{ assume a-in: a ∈ read-only (share (dropWhile (Not ◦ is-volatile-Write_{sb}) sb) S)
assume nv: ¬ volatile
have a ∈ read-only (share sb S_{sb})
proof (cases a ∈ O_{sb} ∪ all-acquired sb)
case True
  from share-all-until-volatile-write-thread-local' [OF ownership-distinct-ts_{sb}
    sharing-consis-ts_{sb} i-bound ts_{sb}-i True] True a-in
  show ?thesis
    by (simp add: S read-only-def)
next
case False
  from read-only-share-unowned [OF weak-consis-drop - a-in] False
    acquired-all-acquired [of True ?take-sb O_{sb}] all-acquired-append [of ?take-sb
    ?drop-sb]
  have a-ro-shared: a ∈ read-only S
    by auto
  have a /∈ ∪ ((λ(−, −, −, sb, −, −, −).
    all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb)) ' set ts_{sb})
proof −
  { fix k p_k is_k θ_k sb_k D_k O_k R_k
    assume k-bound: k < length ts_{sb}
    assume ts-k: ts_{sb} ! k = (p_k,is_k,θ_k,\emptyset, D_k, O_k, R_k)
    assume a-in: a ∈ all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_k)
    have False
      proof (cases k=1)
        case True with False ts_{sb}-i ts-k a-in
          all-shared-acquired-or-owned [OF sharing-consis [OF k-bound ts-k]]
          all-shared-append [of takeWhile (Not ◦ is-volatile-Write_{sb}) sb_k
            dropWhile (Not ◦ is-volatile-Write_{sb}) sb_k] show False by auto
        next
case False
from rels-nv-cond [rule-format, OF nv k-bound [simplified leq] False [symmetric]
ts-sim [rule-format, OF k-bound] ts-k

\begin{align*}
\text{have } a \notin \text{dom (release (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) (dom (S_{sb})))} \\
\mathcal{R}_k \end{align*}

by (auto simp add: Let-def)

with dom-release-takeWhile [of sb \(k\) (dom (S_{sb})) \mathcal{R}_k]

obtain

\begin{align*}
a\text{-rels}; a \notin \text{dom } \mathcal{R}_k \text{ and} \\
a\text{-shared}; a \notin \text{all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) }
\end{align*}

by auto

with False a-in show \(?\)thesis

by auto

qed

\text{thus } \(?\)thesis by (fastforce simp add: in-set-conv-nth)

qed

with read-only-shared-all-until-volatile-write-subset’ [of ts_{sb} S_{sb}] a-ro-shared

have a \in \text{read-only } S_{sb}

by (auto simp add: S)

from read-only-share-unowned’ [OF weak-consis-sb read-only-unowned [OF i-bound ts_{sb}-i] False this]

show \(?\)thesis .

qed

\text{note non-vol-ro-reduction = this}

have valid-own’: valid-ownership S_{sb}’ ts_{sb}’

proof (intro-locales)

show outstanding-non-volatile.refs-owned-or-read-only S_{sb}’ ts_{sb}’

proof (cases volatile)

\begin{align*}
\text{case False} \\
\text{from outstanding-non-volatile.refs-owned-or-read-only [OF i-bound ts_{sb}-i]}
\end{align*}

have non-volatile-owned-or-read-only False S_{sb} O_{sb} sb.

then

\begin{align*}
\text{have non-volatile-owned-or-read-only False S_{sb} O_{sb} (sb@[Read_{sb} False a t v])} \\
\text{using access-cond’ False non-vol-ro-reduction}
\end{align*}

by (auto simp add: non-volatile-owned-or-read-only-append)

from outstanding-non-volatile.refs-owned-or-read-only-nth-update [OF i-bound this]

show \(?\)thesis by (auto simp add: False ts_{sb}’ sb’ O_{sb}’ S_{sb}’)

next

\begin{align*}
\text{case True} \\
\text{from outstanding-non-volatile.refs-owned-or-read-only [OF i-bound ts_{sb}-i]}
\end{align*}

have non-volatile-owned-or-read-only False S_{sb} O_{sb} sb.

then

\begin{align*}
\text{have non-volatile-owned-or-read-only False S_{sb} O_{sb} (sb@[Read_{sb} True a t v])} \\
\text{using True}
\end{align*}

by (simp add: non-volatile-owned-or-read-only-append)

from outstanding-non-volatile.refs-owned-or-read-only-nth-update [OF i-bound this]

show \(?\)thesis by (auto simp add: True ts_{sb}’ sb’ O_{sb}’ S_{sb}’)

qed
show outstanding-volatile-writes-unowned-by-others $ts_{sb}'$
proof
  have out: outstanding-refs is-volatile-Write$_{sb}$ (sb @ [Read$_{sb}$ volatile a t v]) $\subseteq$ outstanding-refs is-volatile-Write$_{sb}$ sb
    by (auto simp add: outstanding-refs-append)
  have all-acquired (sb @ [Read$_{sb}$ volatile a t v]) $\subseteq$ all-acquired sb
    by (auto simp add: all-acquired-append)
  from outstanding-volatile-writes-unowned-by-others-store-buffer [OF i-bound $ts_{sb}$-i out this]
  show ?thesis by (simp add: $ts_{sb}$ $sb$ $sb$' $O_{sb}'$
qed

next
show read-only-reads-unowned $ts_{sb}'$
proof (cases volatile)
case True
  have r: read-only-reads (acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) (sb @ [Read$_{sb}$ volatile a t v])) $O_{sb}$)
    (dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) (sb @ [Read$_{sb}$ volatile a t v]))
    $\subseteq$ read-only-reads (acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $O_{sb}$)
  apply (case-tac outstanding-refs (is-volatile-Write$_{sb}$) sb = {}) 
  apply (simp-all add: outstanding-vol-write-take-drop-appends acquired-append read-only-reads-append True)
done

  have $O_{sb}$ $\cup$ all-acquired (sb @ [Read$_{sb}$ volatile a t v]) $\subseteq$ $O_{sb}$ $\cup$ all-acquired sb
    by (simp add: all-acquired-append)
from read-only-reads-unowned-nth-update [OF i-bound $ts_{sb}$-i r this]
show ?thesis by (simp add: $ts_{sb}$' $O_{sb}'$ $sb'$)
next
case False
proof (unfold-locales)
  fix n m
  fix $p_n$, $is_n$, $O_n$, $R_n$, $D_n$, $\vartheta_n$, sb$_n$, $p_m$, $is_m$, $O_m$, $R_m$, $D_m$, $\vartheta_m$, sb$_m$
  assume n-bound: $n < \text{length } ts_{sb}'$
  and m-bound: $m < \text{length } ts_{sb}'$
  and neq-n-m: $n \neq m$
  and nth: $ts_{sb}'!n = (p_n, is_n, \vartheta_n, sb_n, D_n, O_n, R_n)$
  and nth: $ts_{sb}'!m = (p_m, is_m, \vartheta_m, sb_m, D_m, O_m, R_m)$
  from n-bound have n-bound': $n < \text{length } ts_{sb}$ by (simp add: $ts_{sb}$')
  from m-bound have m-bound': $m < \text{length } ts_{sb}$ by (simp add: $ts_{sb}$')
  have acq-eq: $(O_{sb}' \cup \text{all-acquired } sb') = (O_{sb} \cup \text{all-acquired } sb)$
    by (simp add: all-acquired-append sb' $O_{sb}'$)

  next
  from read-only-reads-unowned-nth-update [OF i-bound $ts_{sb}$-i r this]
  show ?thesis by (simp add: $ts_{sb}$' $O_{sb}'$ $sb'$)

  next
  case False
  proof (unfold-locales)
    fix n m
    fix $p_n$, $is_n$, $O_n$, $R_n$, $D_n$, $\vartheta_n$, sb$_n$, $p_m$, $is_m$, $O_m$, $R_m$, $D_m$, $\vartheta_m$, sb$_m$
    assume n-bound: $n < \text{length } ts_{sb}'$
    and m-bound: $m < \text{length } ts_{sb}'$
    and neq-n-m: $n \neq m$
    and nth: $ts_{sb}'!n = (p_n, is_n, \vartheta_n, sb_n, D_n, O_n, R_n)$
    and nth: $ts_{sb}'!m = (p_m, is_m, \vartheta_m, sb_m, D_m, O_m, R_m)$
    from n-bound have n-bound': $n < \text{length } ts_{sb}$ by (simp add: $ts_{sb}$')
    from m-bound have m-bound': $m < \text{length } ts_{sb}$ by (simp add: $ts_{sb}$')
    have acq-eq: $(O_{sb}' \cup \text{all-acquired } sb') = (O_{sb} \cup \text{all-acquired } sb)$
      by (simp add: all-acquired-append sb' $O_{sb}'$)

    445
show \((\mathcal{O}_m \cup \text{all-acquired } \mathcal{S}_b) \cap \text{read-only-reads} \ (\text{acquired } \text{True} \ (\text{takeWhile } (\text{Not} \circ \text{is-volatile-Write}_s) \mathcal{S}_b) \mathcal{O}_n) \ (\text{dropWhile } (\text{Not} \circ \text{is-volatile-Write}_s) \mathcal{S}_b) = \{\} \) 

proof (cases \(m=\text{\#i}\)) 
  case True 
  with \(\text{\#eq-n-m}\) have \(\text{\#neq-n-i}\): \(n \neq i\) by auto 
  note read-only-reads-unowned \([\text{OF } \text{n-bound' } \text{i-bound } \text{\#nth } \text{\#ts}\_\text{sb}'\] 
  moreover 
  note acq-eq 
  ultimately show \(?\text{thesis}\) 
  using True \(\text{ts}\_\text{sb}'\_\text{i nth m-bound'} \text{\#m-bound}'\) 
  by \((\text{simp add: } \text{ts}_\text{sb}'\) 

next 
  case False 
  note \(\text{\#neq-m-i = this}\) 
  with \(\text{m-bound mth i-bound}\) have \(\text{\#mth}': \text{ts}_\text{sb}'!m = (p_\text{m}, is_\text{m}, 0_\text{m}, sb_\text{m}, D_\text{m}, \mathcal{O}_m, \mathcal{R}_m)\) 
  by \((\text{auto simp add: } \text{ts}_\text{sb}'\) 
  show \(?\text{thesis}\) 
  proof (cases \(n=\text{\#i}\)) 

next 
  case False 
  with \(\text{\#n-bound nth i-bound}\) have \(\text{\#nth}' : \text{ts}_\text{sb}'!n = (p_n, is_n, 0_n, sb_n, D_n, \mathcal{O}_n, \mathcal{R}_n)\) 
  by \((\text{auto simp add: } \text{ts}_\text{sb}'\) 
  from read-only-reads-unowned \([\text{OF } \text{i-bound m-bound'} \text{\#neq-m-i [symmetric]} \text{ ts}_\text{sb}' \text{\#mth}'\] 
  moreover 
  note acq-eq 
  moreover 
  note non-volatile-unowned-others \([\text{OF } \text{m-bound'} \text{\#neq-m-i [symmetric]} \text{\#mth}'\] 
  ultimately show \(?\text{thesis}\) 
  using True \(\text{ts}_\text{sb}'\_\text{i nth m-bound'} \text{\#neq-m-i}\) 
  apply \((\text{case-tac outstanding-refs (is-volatile-Write}_s) \mathcal{S}_b = \{\})\) 
  apply \((\text{clarsimp simp add: outstanding-vol-write-take-drop-appends})\) 
  acquired-append read-only-reads-append \(\text{ts}_\text{sb}' \text{\#sb}' \text{\#\mathcal{O}_s}'\) 
  done 

next 
  case False 
  with \(\text{\#n-bound nth i-bound}\) have \(\text{\#nth}' : \text{ts}_\text{sb}'!n = (p_n, is_n, 0_n, sb_n, D_n, \mathcal{O}_n, \mathcal{R}_n)\) 
  by \((\text{auto simp add: } \text{ts}_\text{sb}'\) 
  from read-only-reads-unowned \([\text{OF } \text{m-bound'} \text{\#m-bound'} \text{\#neq-n-m nth'mth'}\] False \(\text{\#neq-m-i}\) 
  show \(?\text{thesis}\) 
  by \((\text{clarsimp})\) 
  qed 
  qed 
  qed 
  next 
  show ownership-distinct \(\text{ts}_\text{sb}'\) 

446
proof
  -
  have all-acquired (sb @ [Read\sb volatile a t v]) ⊆ all-acquired sb
    by (auto simp add: all-acquired-append)
  from ownership-distinct-instructions-read-value-store-buffer-independent
    [OF i-bound ts\sb-i this]
  show ?thesis by (simp add: ts\sb' sb' \O\sb')
qed

have valid-hist': valid-history program-step ts\sb'
proof
  -
  from valid-history [OF i-bound ts\sb-i]
  have hcons: history-consistent \tho sb (hd-prog p\sb sb) sb.
  from load-tmps-read-tmps-distinct [OF i-bound ts\sb-i]
  have t-notin-reads: t ∉ read-tmps sb
    by (auto simp add: is sb)
  from load-tmps-write-tmps-distinct [OF i-bound ts\sb-i]
  have t-notin-writes: t ∉ (∪ (fst ' write-sops sb))
    by (auto simp add: is sb)
  have valid-sops: ∀ sop ∈ write-sops sb. valid-sop sop
    by auto
  from valid-write-sops [OF i-bound ts\sb-i]
  have valid-sops: ∀ sop ∈ write-sops sb. valid-sop sop
    by auto
  from load-tmps-fresh [OF i-bound ts\sb-i]
  have t-fresh: t ∉ dom \tho sb
    using is sb
    by simp
  have history-consistent (\tho sb (t↦→ v))
    (hd-prog p\sb (sb@ [Read\sb volatile a t v])) (sb@ [Read\sb volatile a t v])
    using t-notin-writes valid-sops t-fresh hcons
    valid-implies-valid-prog-hd [OF i-bound ts\sb-i valid]
    apply
    apply (rule history-consistent-appendI)
    apply (auto simp add: hd-prog-append-Readsb)
    done
  from valid-history-nth-update [OF i-bound this]
  show ?thesis
    by (auto simp add: ts\sb' sb' \O\sb' \tho sb')
qed

from reads-consistent-buffered-snoc [OF buf-v valid-reads [OF i-bound ts\sb-i]
  volatile-cond]
  have reads-consis': reads-consistent False \O\sb m\sb (sb @ [Read\sb volatile a t v])
    by (simp split: if-split-asm)

from valid-reads-nth-update [OF i-bound this]
have valid-reads': valid-reads m\sb ts\sb' by (simp add: ts\sb' sb' \O\sb')

have valid-sharing': valid-sharing \S\sb' ts\sb'
proof (intro-locales)

from outstanding-non-volatile-writes-unshared [OF i-bound ts_{sb} -i]
have non-volatile-writes-unshared S_{sb} (sb @ [Read_{sb} volatile a t v])
  by (auto simp add: non-volatile-writes-unshared-append)

from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared S_{sb}' ts_{sb}'
  by (simp add: ts_{sb}' sb' S_{sb}')

next

from sharing-consis [OF i-bound ts_{sb} -i]
have sharing-consistent S_{sb} O_{sb} sb.

then

have sharing-consistent S_{sb} O_{sb} (sb @ [Read_{sb} volatile a t v])
  by (simp add: sharing-consistent-append)

from sharing-consis-nth-update [OF i-bound this]
show sharing-consis S_{sb}' ts_{sb}'
  by (simp add: ts_{sb}' O_{sb}' sb' S_{sb}')

next

note read-only-unowned [OF i-bound ts_{sb} -i]

from read-only-unowned-nth-update [OF i-bound this]
show read-only-unowned S_{sb}' ts_{sb}'
  by (simp add: S_{sb}' ts_{sb}' sb' O_{sb}')

next

from unowned-shared-nth-update [OF i-bound ts_{sb} -i subset-refl]
show unowned-shared S_{sb}' ts_{sb}'
by (simp add: ts_{sb}' O_{sb}' S_{sb}')

next

from no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb} -i]
have no-write-to-read-only-memory S_{sb} sb.

hence no-write-to-read-only-memory S_{sb} (sb @ [Read_{sb} volatile a t v])
  by (simp add: no-write-to-read-only-memory-append)

from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
show no-outstanding-write-to-read-only-memory S_{sb}' ts_{sb}'
  by (simp add: ts_{sb}' S_{sb}' sb')

qed

have tmqs-distinct ts_{sb}'
  proof (intro-locales)

from load-tmps-distinct [OF i-bound ts_{sb} -i]
have distinct-load-tmps is_{sb}'
  by (auto split: instr.splits simp add: is_{sb})

from load-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-distinct ts_{sb}'
by (simp add: ts_{sb}')

next

from read-tmps-distinct [OF i-bound ts_{sb} -i]
have distinct-read-tmps sb.

moreover

from load-tmps-read-tmps-distinct [OF i-bound ts_{sb} -i]
have t \notin read-tmps sb
  by (auto simp add: is_{sb})

ultimately have distinct-read-tmps (sb @ [Read_{sb} volatile a t v])
  by (auto simp add: distinct-read-tmps-append)
from \texttt{read-tmps-distinct-nth-update} [OF i-bound this]

\textbf{show} \texttt{read-tmps-distinct ts}\_sb\,' \textbf{by} (simp add: ts\_sb\,' sb\')

\textbf{next}

\textbf{from} \texttt{load-tmps-read-tmps-distinct} [OF i-bound ts\_sb\,-i]

\textbf{load-tmps-distinct} [OF i-bound ts\_sb\,-i]

\textbf{have} \texttt{load-tmps is}\_sb\,' \cap \texttt{read-tmps} (sb @ [Read\_sb\ volatile a t v]) = \{\}

\textbf{by} (clarsimp simp add: read-tmps-append is\_sb)

\textbf{from} \texttt{load-tmps-read-tmps-distinct-nth-update} [OF i-bound this]

\textbf{show} \texttt{load-tmps-read-tmps-distinct ts}\_sb\,' \textbf{by} (simp add: ts\_sb\,' sb\')

\textbf{qed}

\textbf{have} valid-sops\': valid-sops ts\_sb\,'

\textbf{proof} –

\textbf{from} \texttt{valid-store-sops} [OF i-bound ts\_sb\,-i]

\textbf{have} valid-store-sops\': \(\forall\) sop\(\in\)store-sops is\_sb\,' valid-sop sop

\textbf{by} (auto simp add: is\_sb)

\textbf{from} \texttt{valid-write-sops} [OF i-bound ts\_sb\,-i]

\textbf{have} valid-write-sops\': \(\forall\) sop\(\in\)write-sops (sb@ [Read\_sb\ volatile a t v]). valid-sop sop

\textbf{by} (auto simp add: write-sops-append)

\textbf{from} \texttt{valid-sops-nth-update} [OF i-bound valid-write-sops\' valid-store-sops\']

\textbf{show} \?thesis \textbf{by} (simp add: ts\_sb\,' sb\')

\textbf{qed}

\textbf{have} valid-dd\': valid-data-dependency ts\_sb\,'

\textbf{proof} –

\textbf{from} \texttt{data-dependency-consistent-instrs} [OF i-bound ts\_sb\,-i]

\textbf{have} dd-is: data-dependency-consistent-instrs (dom \(\theta\) sb\,' is\_sb\,' )

\textbf{by} (auto simp add: is\_sb \(\theta\) sb\,')

\textbf{from} \texttt{load-tmps-write-tmps-distinct} [OF i-bound ts\_sb\,-i]

\textbf{have} load-tmps is\_sb\,' \cap \bigcup (fst ' write-sops (sb@ [Read\_sb\ volatile a t v])) = \{\}

\textbf{by} (auto simp add: write-sops-append is\_sb)

\textbf{from} \texttt{valid-data-dependency-nth-update} [OF i-bound dd-is this]

\textbf{show} \?thesis \textbf{by} (simp add: ts\_sb\,' sb\')

\textbf{qed}

\textbf{have} load-tmps-fresh\': load-tmps-fresh ts\_sb\,'

\textbf{proof} –

\textbf{from} \texttt{load-tmps-fresh} [OF i-bound ts\_sb\,-i]

\textbf{have} load-tmps (Read volatile a t # is\_sb\,' ) \cap dom \(\theta\) sb = \{\}

\textbf{by} (simp add: is\_sb)

\textbf{moreover}

\textbf{from} \texttt{load-tmps-distinct} [OF i-bound ts\_sb\,-i] \textbf{have} t \(\notin\) load-tmps is\_sb\,'

\textbf{by} (auto simp add: is\_sb)

\textbf{ultimately have} load-tmps is\_sb\,' \cap dom (\(\bar{\theta}\) sb (t \(\mapsto\) v)) = \{\}

\textbf{by} auto

\textbf{from} \texttt{load-tmps-fresh-nth-update} [OF i-bound this]

\textbf{show} \?thesis \textbf{by} (simp add: ts\_sb\,' sb\' \(\bar{\theta}\) sb\')

\textbf{qed}

\textbf{have} enough-flushs\': enough-flushs ts\_sb\,'

449
proof 

from clean-no-outstanding-volatile-Write sb [OF i-bound ts sb-i]
have \( \neg D_{sb} \rightarrow \) outstanding-ref{s} is-volatile-Write sb \((sb@[Read_{sb}\text{ volatile a t v}) = {} \)
by (auto simp add: outstanding-ref{s}-append )
from enough-flushs-nth-update [OF i-bound this]
show \(?\)thesis
by (simp add: ts sb′)
qed

have valid-program-history′: valid-program-history ts sb′
proof 

from valid-program-history [OF i-bound ts sb-i]
have causal′: causal-program-history is sb sb′.
then have causal′: causal-program-history is sb′ sb (sb@[Read_{sb}\text{ volatile a t v}]) = is sb'
by (auto simp: causal-program-history-Read is sb)
from valid-last-prog [OF i-bound ts sb-i]
have last-prog p sb = p sb.
then have causal′: causal-program-history is sb′ sb (sb@[Read_{sb}\text{ volatile a t v}]) = p sb
by (simp add: last-prog-append-Read sb)

from valid-program-history-nth-update [OF i-bound causal′ this]
show \(?\)thesis
by (simp add: ts sb′)
qed

from True have flush-all: takeWhile (Not ◦ is-volatile-Write sb) sb = sb
by (auto simp add: outstanding-ref{s}-conv )

from True have suspend-nothing: dropWhile (Not ◦ is-volatile-Write sb) sb = []
by (auto simp add: outstanding-ref{s}-conv)

hence suspends-empty: suspends = []
by (simp add: suspends)
from suspends-empty is-sim have: is = Read volatile a t # is sb′
by (simp add: is sb)
with suspends-empty ts-i
have ts-i: ts!i = (p_{sb}, Read volatile a t # is sb′, \( \emptyset_{sb},() \), D, acquired True \?take-sb O sb,
release \?take-sb (dom S sb) R sb)
by simp

from direct-memop-step.Read
have (Read volatile a t # is sb′, \( \emptyset_{sb},() \), m, D, acquired True \?take-sb O sb,
release \?take-sb (dom S sb) R sb, S) \rightarrow
(is sb′, \( \emptyset_{sb}(t \mapsto m a),() \), m, D, acquired True \?take-sb O sb,release \?take-sb (dom S sb) R sb, S),
from direct-computation.concurrent-step.Memop [OF i-bound′ ts-i this]
have (ts, m, S) \rightarrow_d (ts[i := (p_{sb}, is sb′, \( \emptyset_{sb}(t \mapsto m a),() \),)
\( D, \) acquired True \(?\text{take-sb} \mathcal{O}_{sb}, \) release \(?\text{take-sb} (\text{dom} \mathcal{S}_{sb}) \mathcal{R}_{sb})\], \( m, \mathcal{S} \).

moreover

from \( \text{flush-all-until-volatile-write-Read-commute} \) \{OF \text{i-bound} \text{ts}_{sb}\cdot i \} \) \{simplified \text{is}_{sb} \}

have \text{flush-commute}: \text{flush-all-until-volatile-write}

\( (\text{ts}_{sb}\cdot i := (p_{sb}, \text{i}^{	ext{ts}}_{sb}, \vartheta_{sb}(t \mapsto v)), \text{sb} @ [\text{Read}_{sb} \text{volatile} \ a \ t \ v], \mathcal{D}_{sb}, \mathcal{O}_{sb}, \mathcal{R}_{sb})]) \) \( \text{m}_{sb} = \text{flush-all-until-volatile-write} \text{ts}_{sb} \text{m}_{sb}. \)

from \( \text{True} \) witness \text{have not-volatile}' \( \text{volatile}' = \text{False} \)

by \( \) \{auto \text{simp add: outstanding-refs-conv} \}

from witness not-volatile' \text{have a-out-sb}: \text{a} \in \text{outstanding-refs} \ (\text{Not} \circ \text{is-volatile}) \text{sb}

apply \{\text{cases} \text{sop}'\}

apply \{\text{fastforce simp add: outstanding-refs-conv is-volatile-def split: memref.splits} \}

done

with \( \text{non-volatile-owned-or-read-only-outstanding-refs} \)
\{OF \text{outstanding-non-volatile-refs-owned-or-read-only} \} \{OF \text{i-bound} \text{ts}_{sb}\cdot i \}

have \text{a-owned}: \text{a} \in \mathcal{O}_{sb} \cup \text{all-acquired} \text{sb} \cup \text{read-only-reads} \mathcal{O}_{sb} \text{sb}

by \( \) \{auto \}

have \text{flush-all-until-volatile-write} \text{ts}_{sb} \text{m}_{sb} \text{a} = v

proof −

have \( \forall \ j < \) \text{length} \text{ts}_{sb}. \ i \neq j \rightarrow

\( (\text{let} (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) = \text{ts}_{sb}\cdot j \) \)

\( \text{in} \ a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb} \text{sb}_{j})) \)

proof −

\{ \}

fix \( j \ p_{j} \text{i}_{j} \mathcal{O}_{j} \mathcal{R}_{j} \mathcal{D}_{j} \text{xs}_{j} \text{sb}_{j} \)

assume \( \text{j-bound} : j < \) \text{length} \text{ts}_{sb}

assume \( \text{neq-i-j} : i \neq j \)

assume \( \text{jth} : \text{ts}_{sb}\cdot j = (p_{j}, \text{i}_{j}, \text{xs}_{j}, \text{sb}_{j}, \mathcal{D}_{j}, \mathcal{O}_{j}, \mathcal{R}_{j}) \)

have \( a \notin \) \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb} \text{sb}_{j})) \)

proof −

let \( \text{?take-sb}_{j} = (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb} \text{sb}_{j})) \)

let \( \text{?drop-sb}_{j} = (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb} \text{sb}_{j})) \)

assume \( \text{a-in} : a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \text{sb}_{j} \)

with \( \text{outstanding-refs-takeWhile} \) \{\text{where} \ P' = \text{Not} \circ \text{is-volatile-Write}_{sb} \}

have \( \text{a-in}' : a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \text{sb}_{j} \)

by \( \) \{auto \}

with \( \text{non-volatile-owned-or-read-only-outstanding-non-volatile-writes} \)
\{OF \text{outstanding-non-volatile-refs-owned-or-read-only} \} \{OF \text{i-bound} \text{jth} \}

have \( \text{j-owns} : a \in \mathcal{O}_{j} \cup \text{all-acquired} \text{sb}_{j} \)

by \( \) \{auto \}

with \( \text{ownership-distinct} \) \{OF \text{i-bound} \text{j-bound} \text{neq-i-j ts}_{sb}\cdot i \text{ jth} \}

451
have a-not-owns: $a \notin \mathcal{O}_{sb} \cup$ all-acquired $sb$
by blast

from non-volatile-owned-or-read-only-append [of False $S_{sb}$ $O_j$ ?take-sb$_j$ ?drop-sb$_j$]
outstanding-non-volatile-refs-owned-or-read-only [OF $j$-bound $j$th]
have non-volatile-owned-or-read-only False $S_{sb}$ $O_j$ ?take-sb$_j$
by simp
from non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF this] a-in
have j-owns-drop: $a \in \mathcal{O}_j \cup$ all-acquired ?take-sb$_j$
by auto

from rels-cond [rule-format, OF $j$-bound [simplified leq neq-i-j] ts-sim
[rule-format, OF $j$-bound] $j$th
have no-unsharing:release ?take-sb$_j$ (dom ($S_{sb}$)) $R_j$ $a \neq$ Some False
by (auto simp add: Let-def)

{ 
  assume a \in acquired True sb $\mathcal{O}_{sb}$
  with acquired-all-acquired-in [OF this] ownership-distinct [OF $i$-bound $j$-bound neq-i-j
  $ts_{sb}$-i $j$th]
  j-owns
  have False
  by auto
}
moreover
{
  assume a-share: volatile $\land$ $a \in \text{dom (share ?drop-sb $S$)}$

  from read-only-share-unowned-in [OF weak-consis-drop a-ro] a-not-owns
  acquired-all-acquired [of True ?take-sb $\mathcal{O}_{sb}$]
  all-acquired-append [of ?take-sb ?drop-sb]
  have $a \in \text{read-only } S$
  by auto
  with share-all-until-volatile-write-thread-local [OF ownership-distinct-ts$_{sb}$
  sharing-consis-ts$_{sb}$ $j$-bound $j$th $j$-owns]
  have $a \in \text{read-only (share ?take-sb$_j$ $S_{sb}$)}$
  by (auto simp add: read-only-def $S$)
  hence a-dom: $a \in \text{dom (share ?take-sb$_j$ $S_{sb}$)}$
  by (auto simp add: read-only-def domIff)
  from outstanding-non-volatile-writes-unshared [OF $j$-bound $j$th]
  non-volatile-writes-unshared-append [of $S_{sb}$ ?take-sb$_j$ ?drop-sb$_j$]
  have nvw: non-volatile-writes-unshared $S_{sb}$ ?take-sb$_j$ by auto
  from release-not-unshared-no-write-take [OF this no-unsharing a-dom] a-in
  have False by auto
}
moreover
{
  assume a-share: volatile $\land$ $a \in \text{dom (share ?drop-sb $S$)}$
  from outstanding-non-volatile-writes-unshared [OF $j$-bound $j$th]
have non-volatile-writes-unshared $S_{sb}$.
with non-volatile-writes-unshared-append [of $S_{sb}$ ?take-sb ?drop-sb]
    have unshared-take: non-volatile-writes-unshared $S_{sb}$ (takeWhile (Not o is-volatile-Write$_{sb}$) sb)
        by clarsimp
from valid-own have own-dist: ownership-distinct ts$_{sb}$
    by (simp add: valid-ownership-def)
from valid-sharing have sharing-consis $S_{sb}$ ts$_{sb}$
    by (simp add: valid-sharing-def)
from sharing-consistent-share-all-until-volatile-write [OF own-dist this i-bound ts$_{sb}$-i]
have sc: sharing-consistent $S$ (acquired True ?take-sb O$_{sb}$) ?drop-sb
    by (simp add: S)
from sharing-consistent-share-all-shared
have dom (share ?drop-sb $S$) ⊆ dom $S$ ∪ all-shared ?drop-sb
    by auto
also from sharing-consistent-all-shared [OF sc]
have . . . ⊆ dom $S$ ∪ acquired True ?take-sb O$_{sb}$ by auto
also from acquired-all-acquired all-acquired-takeWhile
have . . . ⊆ dom $S$ ∪ (O$_{sb}$ ∪ all-acquired sb) by force
finally
have a-shared: a ∈ dom $S$
    using a-share a-not-owns
    by auto

    with share-all-until-volatile-write-thread-local [OF ownership-distinct-ts$_{sb}$ sharing-consis-ts$_{sb}$ j-bound jth j-owns]
    have a-dom: a ∈ dom (share ?take-sb $S_{sb}$)
        by (auto simp add: S domIff)
    from release-not-unshared-no-write-take [OF unshared-take no-unsharing a-dom] a-in
    have False by auto

} ultimately show False
    using access-cond’
    by auto
    qed

} thus ?thesis
    by (fastforce simp add: Let-def)
qed

from flush-all-until-volatile-write-buffered-val-conv
[OF True i-bound ts$_{sb}$-i this]
show ?thesis
    by (simp add: buf-v)
qed

453
hence \( m \cdot a = v \)
by (simp add: m)

have tmps-commute: \( \emptyset_{sb}(t \mapsto v) = (\emptyset_{sb} \upharpoonright (\text{dom } \emptyset_{sb} - \{t\}))(t \mapsto v) \)
apply (rule ext)
apply (auto simp add: restrict_map_def domIff)
done

from suspend-nothing
have suspend-nothing \': (\text{dropWhile} \ (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb') = []
by (simp add: sb')

from \( D \)
have \( D' \): \( D' = (D \cup \text{outstanding-refs}\ \text{is-volatile-Write}_{sb} \ (sb@[\text{Read}_{sb} \text{volatile} a t v])) \neq {} \)
by (auto simp: outstanding-refs-append)

have \( (ts_{sb}',m_{sb},S_{sb}') \sim (ts[\ i := (p_{sb},is_{sb}',\emptyset_{sb}'sb',D_{sb}'R_{sb}'),m,S]) \)
apply (rule sim-config.intros)
apply (simp add: m flush-commute ts_{sb}'O_{sb}'\emptyset_{sb}'sb'D_{sb}'R_{sb}')
using share-all-until-volatile-write-Read-commute [OF i-bound ts_{sb}-i [simplified is_{sb}]]
apply (simp add: S S_{sb}'ts_{sb}'sb'O_{sb}'\emptyset_{sb}'R_{sb}')
using leq
apply (simp add: ts_{sb}'o)
using i-bound i-bound' ts-sim ts-i True \( D' \)
apply (clarsimp simp add: Let-def nth-list-update
outstanding-refs-conv m-a-v ts_{sb}'O_{sb}'S_{sb}'\emptyset_{sb}'sb'R_{sb}'suspend-nothing'
\( D_{sb}'\) flush-all acquired-append release-append
split: if-split-asm )
apply (rule tmps-commute)
done

ultimately show \( ?\text{thesis} \)
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct'
valid-sops' valid-dd' load-tmps-fresh' enough-flushs'
valid-program-history' valid'
m_{sb}'S_{sb}'O_{sb}'
by (auto simp del: fun-upd-apply )

next
case False

then obtain \( r \ where \ r\in \text{set } sb \) and volatile-r: is-volatile-Write_{sb} \ r
by (auto simp add: outstanding-refs-conv)
from takeWhile-dropWhile-real-prefix
[OF r-in, of (\text{Not } \circ \text{is-volatile-Write}_{sb}), simplified, OF volatile-r]
obtain \( a'v'sb''\ sop' A'L'R'W' \ where \)
sb-split: \( \text{sb} = \text{takeWhile} \left( \neg \text{is-volatile-Write}_\text{sb} \right) \text{sb} @ \text{Write}_\text{sb} \) True a' sop' v' A' L' R' W'\# \text{sb}''

and

drop: \( \text{dropWhile} \left( \neg \text{is-volatile-Write}_\text{sb} \right) \text{sb} = \text{Write}_\text{sb} \) True a' sop' v' A' L' R' W'\# \text{sb}''

\( \text{apply} \) (auto)

\( \text{subgoal for y ys} \)

\( \text{apply} \) (case-tac y)

\( \text{apply} \) auto

\( \text{done} \)

\( \text{done} \)

from drop suspends have suspends: suspends = \( \text{Write}_\text{sb} \) True a' sop' v' A' L' R' W'\# \text{sb}''

by simp

have \( (\text{ts, m, } \mathcal{S}) \Rightarrow^*_d (\text{ts, m, } \mathcal{S}) \) by auto

moreover

from flush-all-until-volatile-write-Read-commute [OF i-bound ts_{\text{sb}}-i

[ simplified is_{\text{sb}} ] ]

have flush-commute: flush-all-until-volatile-write

\( (\text{ts}_{\text{sb}}[i := (p_{\text{sb}}, \text{is}_{\text{sb}}', \vartheta_{\text{sb}}(t \mapsto v), \text{sb} @ [\text{Read}_{\text{sb}} \text{volatile a t v}], \mathcal{D}_{\text{sb}}, \mathcal{O}_{\text{sb}}, \mathcal{R}_{\text{sb}})]) \)

\( m_{\text{sb}} = \)

flush-all-until-volatile-write ts_{\text{sb}} m_{\text{sb}}.

have \( \text{Write}_{\text{sb}} \) True a' sop' v' A' L' R' W'\in \text{set sb}

by (subst sb-split) auto

from dropWhile-append1 [OF this, of (Not o is-volatile-Write_{\text{sb}})]

have drop-app-comm:

\( (\text{dropWhile} (\neg \text{is-volatile-Write}_{\text{sb}}) (\text{sb} @ [\text{Read}_{\text{sb}} \text{volatile a t v}])) = \)

\( \text{dropWhile} (\neg \text{is-volatile-Write}_{\text{sb}}) \) \text{sb} @ [\text{Read}_{\text{sb}} \text{volatile a t v}]

by simp

from load-tmps-fresh [OF i-bound ts_{\text{sb}}-i]

have \( t \notin \text{dom } \varnothing_{\text{sb}} \)

by (auto simp add: is_{\text{sb}})

then have tmps-commute:

\( \varnothing_{\text{sb}} \mid (\text{dom } \varnothing_{\text{sb}} - \text{read-tmps sb}''') = \)

\( \varnothing_{\text{sb}} \mid (\text{dom } \varnothing_{\text{sb}} - \text{insert t (read-tmps sb'''}) \)

apply

apply (rule ext)

apply auto

done

from \( \mathcal{D} \)

455
have \( D' : \mathcal{D}_{sb} = (\mathcal{D} \lor \text{outstanding-refs is-volatile-Write}_{sb} (sb@[\text{Read}_{sb} \text{ volatile a t v}]) \neq \{\}) \)
by (auto simp: outstanding-refs-append)

have \( (ts_{sb}',m_{sb},S_{sb}) \sim (ts,m,S) \)
apply (rule sim-config-intros)
apply (simp add: m flush-commute ts_{sb}' O_{sb}' R_{sb}' \theta_{sb}' D_{sb}' )
using share-all-until-volatile-write-Read-commute [OF i-bound ts_{sb}' i [simplified is_{sb}]]
apply (simp add: S S_{sb}' ts_{sb}' sb' O_{sb}' R_{sb}' \theta_{sb}' )
apply leq
using i-bound i-bound
apply (clarsimp simp add: Let-def nth-list-update is-sim drop-app-comm
read-tmps-append suspends prog-instrs-append-Read sb instrs-append-Read sb
hd-prog-append-Read sb
drop is_{sb} ts_{sb}' sb' O_{sb}' R_{sb}' \theta_{sb}' D_{sb}' acquired-append takeWhile-append1 [OF r-in]
volatile-r
split: if-split-asm)
apply (simp add: drop tmps-commute+)
done

ultimately show \(?thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-dd'
valid-sops' load-tmps-fresh' enough-flushs'
valid-program-history' valid' m_{sb}' S_{sb}'
by (auto simp del: fun-upd-apply )
qed
next
  case (SBHReadUnbuffered a volatile t)
  then obtain
  is_{sb}: is_{sb} = \text{Read volatile a t} \neq is_{sb}' and
  O_{sb}': O_{sb} = O_{sb} and
  R_{sb}': R_{sb} = R_{sb} and
  \theta_{sb}': \theta_{sb} = \theta_{sb}(t\rightarrow (m_{sb} a)) and
  sb': sb = sb@[\text{Read}_{sb} \text{ volatile a t} (m_{sb} a)] and
  m_{sb}': m_{sb} = m_{sb} and
  S_{sb}': S_{sb} = S_{sb} and
  D_{sb}': D_{sb} = D_{sb} and
  buf-None: buffered-val sb a = None

  by auto

  from safe-memop-flush-sb [simplified is_{sb}]
  obtain access-cond': a \in \text{acquired True} sb O_{sb} \lor
  a \in \text{read-only (share ?drop-sb S)} \lor \text{(volatile} \land a \in \text{dom (share ?drop-sb S)}) and
  volatile-clean: volatile \rightarrow \neg D_{sb} and
  rels-cond: \forall j < \text{length ts}. i \neq j \rightarrow \text{released (ts!j) a} \neq \text{Some False and}
  rels-nv-cond: \neg \text{volatile} \rightarrow (\forall j < \text{length ts}. i \neq j \rightarrow a \notin \text{dom (released (ts!j))})
  by cases auto

456
from clean-no-outstanding-volatile-Write_{sb} [OF i-bound ts_{sb}-i] volatile-clean
have volatile-cond: volatile ⟷ outstanding-refs is-volatile-Write_{sb} sb = {}

by auto

{ 
fix j p_j is_{sbj} O_j \ R_j D_{sbj} \ θ_{sbj} sb_j
assume j-bound: j < length ts_{sb}
assume neq-i-j: i ≠ j
assume jth: ts_{sb}[j] = (p_j, is_{sbj}, θ_{sbj}, sb_j, D_{sbj}, O_j, R_j)
assume non-vol: ¬ volatile
have a \notin O_j \cup all-acquired sb_j
proof
assume a-j: a \in O_j \cup all-acquired sb_j
let ?take-sb_j = (takeWhile (Not \circ is-volatile-Write_{sb}) sb_j)
let ?drop-sb_j = (dropWhile (Not \circ is-volatile-Write_{sb}) sb_j)

from ts-sim [rule-format, OF j-bound] jth
obtain suspends_{j} is_{j} D_{j} where
suspends_{j}: instrs suspends_{j} @ is_{sbj} \in seq @ prog-intrs suspends_{j} and
D_{j}: D_{sbj} = (D_{j} \lor outstanding-refs is-volatile-Write_{sb} sb_j ≠ {}) and
ts_{j}: ts_{j}[j] = (hd-prog p_j suspends_{j}, is_{j},
\emptyset_{sbj} | (dom \emptyset_{sbj} \or read-tmps suspends_{j}),(),
D_{j}, acquired True ?take-sb_{j} O_j, release ?take-sb_{j} (dom S_{sb}) R_j)
by (auto simp add: Let-def)

from a-j ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i jth]
have a-notin-sb: a \notin O_{sb} \cup all-acquired sb
by auto
with acquired-all-acquired [of True sb O_{sb}]
have a-not-aeq: a \notin acquired True sb O_{sb} by blast
with access-cond' non-vol
have a-ro: a \in read-only (share ?drop-sb S)
by auto
from read-only-share-unowned-in [OF weak-consis-drop a-ro] a-notin-sb
acquired-all-acquired [of True ?take-sb O_{sb}]
al-acquired-append [of ?take-sb ?drop-sb]
have a-ro-shared: a \in read-only S
by auto

from rels-nv-cond [rule-format, OF non-vol j-bound [simplified leq] neq-i-j ts_j
have a \notin dom (release ?take-sb_{j} (dom (S_{sb})) R_j)
by auto
with dom-release-takeWhile [of sb_{j} (dom (S_{sb})) R_j]
obtain
a-rels_{j}: a \notin dom R_j and
a-shared_{j}: a \notin all-shared ?take-sb_{j}
by auto

have a \notin \bigcup ((\lambda (-, -, -, sb, -, -) . \text{all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)}) ^* \\
    \text{set } ts_{sb})
proof -
{
  \text{fix } k \ p_k \ i_{sb} \ \partial_k \ D_k \ O_k \ \mathcal{R}_k
  \text{assume k-bound: } k < \text{length } ts_{sb}
  \text{assume ts-k: } ts_{sb} ! k = (p_k, i_{sb}, \partial_k, D_k, O_k, \mathcal{R}_k)
  \text{assume a-in: } a \in \text{all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)}
  \text{have False}
proof (cases k=j)
  \text{case True with a-shared jth ts-k a-in show False by auto}
next
  \text{case False}
from ownership-distinct [OF j-bound k-bound False [symmetric] jth ts-k] a-j
have a \notin (O_k \cup \text{all-acquired } sb_k) by auto
with all-shared-acquired-or-owned [OF sharing-consis [OF k-bound ts-k]] a-in
show False
using all-acquired-append [of takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb]
  dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_k
all-shared-append [of takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb]
  dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_k] by auto
qed
}
thus ?thesis by (fastforce simp add: in-set-conv-nth)
qed
with a-ro-shared
  read-only-shared-all-until-volatile-write-subset' [of ts_{sb} \mathcal{S}_{sb}]
have a-ro-shared_{sb}: a \in \text{read-only } \mathcal{S}_{sb}
by (auto simp add: \mathcal{S})

with read-only-unowned [OF j-bound jth]
have a-notin-owns-j: a \notin O_j
by auto

have own-dist: ownership-distinct ts_{sb} by fact
have share-consis: sharing-consis \mathcal{S}_{sb} ts_{sb} by fact
from sharing-consistent-share-all-until-volatile-write [OF own-dist share-consis i-bound ts_{sb}-i]
have consis': sharing-consistent \mathcal{S} (acquired True ?take-sb \mathcal{O}_{sb}) ?drop-sb
by (simp add: \mathcal{S})
from share-all-until-volatile-write-thread-local [OF own-dist share-consis j-bound jth a-j] a-ro-shared
have a-ro-take: a \in \text{read-only (share ?take-sb}_j \mathcal{S}_{sb})
by (auto simp add: domIff \mathcal{S} \text{read-only-def})
from sharing-consis [OF j-bound jth]
have sharing-consistent \( S_{sb} O_j sb_j \).

from sharing-consistent-weak-sharing-consistent [OF this]

weak-sharing-consistent-append [of \( O_j ?\text{take-sb}_j ?\text{drop-sb}_j \)]

have weak-consis-drop:weak-sharing-consistent \( O_j ?\text{take-sb}_j \)

by auto

from read-only-share-acquired-all-shared [OF this read-only-unowned [OF j-bound jth] a-ro-take] a-notin-owns-j a-shared_j

have a \( \notin \) all-acquired ?take-sb_j

by auto

with a-j a-notin-owns-j

have a-drop: a \( \in \) all-acquired ?drop-sb_j

using all-acquired-append [of ?take-sb_j ?drop-sb_j]

by simp

from i-bound j-bound leq have j-bound-ts': j < length ?ts'

by auto

note conflict-drop = a-drop [simplified suspends_j [symmetric]]

from split-all-acquired-in [OF conflict-drop]

show False

proof

assume \( \exists \text{sop } a' v y s z A L R W \),

\((\text{suspends}_j = y s @ \text{Write}_{sb} \text{ True } a' \text{ sop } v A L R W# z s) \land a \in A \)

then

obtain a' \( \text{sop' } v' y s z A' L' R' W' \) where

split-suspends_j: suspends_j = y s @ \text{Write}_{sb} \text{ True } a' \text{ sop' } v' A' L' R' W'# z s

(is suspends_j = ?suspends) and

a-A': a \( \in A' \)

by blast

from sharing-consis [OF j-bound jth]

have sharing-consistent \( S_{sb} O_j sb_j \),

then have \( A' \cap R' = \{\} \)

by (simp add: sharing-consistent-append [of - - ?take-sb_j ?drop-sb_j, simplified]
suspends_j [symmetric] split-suspends_j sharing-consistent-append)

from valid-program-history [OF j-bound jth]

have causal-program-history is_{sbj} sb_j,

then have cph: causal-program-history is_{sbj} ?suspends

apply –

apply (rule causal-program-history-suffix [where sb=?take-sb_j ] )

apply (simp only: split-suspends_j [symmetric] suspends_j)

apply (simp add: split-suspends_j)

done

from ts_j neq-i-j j-bound

have ts'j: ?ts's?j = (hd-prog p_j suspends_j, is_j, \( \tilde{v}_{sbj} \) | (dom \( \tilde{v}_{sbj} \) = read-tmps suspends_j),(),

\( D_j \), acquired True ?take-sb_j \( O_j \), release ?take-sb_j (dom \( S_{sb} \) \( R_j \))
by auto
from valid-last-prog [OF j-bound jth] have last-prog: last-prog \( p_j \) \( s_{b_j} = p_j \).

then have lp: last-prog \( p_j \) suspends\( s_j \) = \( p_j \)
apply 
apply (rule last-prog-same-append [where \( s_b=?\text{take-sb}_{j} \)])
apply (simp only: split-suspends\( s_j \) [symmetric] suspends\( s_j \))
apply simp
done

from valid-reads [OF j-bound jth]
have reads-consis-j: reads-consistent False \( \mathcal{O}_j \) \( m_{s_{b_j}} \) \( s_{b_j} \).
from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing \( S_{s_{b_j}} \) ts\( s_{b_j} \) j-bound]
have reads-consis-m-j: reads-consistent True (acquired True \( ?\text{take-sb}_{j} \) \( \mathcal{O}_j \)) \( m \) suspends\( s_j \)
by (simp add: \( m \) suspends\( s_j \))

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound neq-i-j ts\( s_{b_j} \) i
jth]
have outstanding-refs is-Write\( s_{b_j} \) ?drop-sb \( \cap \) outstanding-refs is-non-volatile-Read\( s_{b_j} \)
 suspends\( s_j \) = \{\}
by (simp add: suspends\( s_j \))
from reads-consistent-flush-independent [OF this reads-consis-m-j]
have reads-consis-flush-suspend: reads-consistent True (acquired True \( ?\text{take-sb}_{j} \) \( \mathcal{O}_j \))
(\( \text{flush} \) ?drop-sb \( m \)) suspends\( s_j \).

hence reads-consis-ys: reads-consistent True (acquired True \( ?\text{take-sb}_{j} \) \( \mathcal{O}_j \))
(\( \text{flush} \) ?drop-sb \( m \)) (\( y\$\)\( \text{Write}_{s_{b_j}} \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \))
by (simp add: split-suspends\( s_j \) reads-consistent-append)

from valid-write-sops [OF j-bound jth]
have \( \forall \) sop\( \in\) write-sops (\( ?\text{take-sb}_{j} @\) ?suspends). valid-sop sop
by (simp add: split-suspends\( s_j \) [symmetric] suspends\( s_j \))
then obtain valid-sops-take: \( \forall \) sop\( \in\) write-sops \( ?\text{take-sb}_{j} \). valid-sop sop and valid-sops-drop: \( \forall \) sop\( \in\) write-sops (\( y\$\)\( \text{Write}_{s_{b_j}} \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)). valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound jth]
have distinct-read-tmps (\( ?\text{take-sb}_{j} @\) ?suspends\( s_j \))
by (simp add: split-suspends\( s_j \) [symmetric] suspends\( s_j \))
then obtain read-tmps-take-drop: read-tmps \( ?\text{take-sb}_{j} \) \( \cap \) read-tmps suspends\( s_j \) = \{\} and
distinct-read-tmps-drop: distinct-read-tmps suspends\( s_j \)
apply (simp only: split-suspends\( s_j \) [symmetric] suspends\( s_j \))
apply (simp only: distinct-read-tmps-append)
done
from valid-history [OF j-bound jth]

have h-consis:
  history-consistent \( \theta_{sbj} \) (hd-prog \( p_j \) (?take-sb\(_j\)@suspends\(_j\))) (?take-sb\(_j\)@suspends\(_j\))
apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog \( p_j \) suspends\(_j\)) \( ?\)take-sb\(_j\) = (hd-prog \( p_j \) suspends\(_j\))
proof
  from last-prog have last-prog \( p_j \) (?take-sb\(_j\)@?drop-sb\(_j\)) = \( p_j \)
  by simp
  from last-prog-hd-prog-append' [OF h-consis] this
have (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
moreover
  have last-prog (hd-prog \( p_j \) (?take-sb\(_j\) @ suspends\(_j\))) ?take-sb\(_j\) = last-prog (hd-prog \( p_j \) suspends\(_j\)) ?take-sb\(_j\)
apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
by (rule last-prog-hd-prog-append)
ultimately show \( ?\)thesis
by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
  h-consis] last-prog-hd-prog
have hist-consis': history-consistent \( \theta_{sbj} \) (hd-prog \( p_j \) suspends\(_j\)) suspends\(_j\)
  by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-ys]
have no-vol-read: outstanding-refs is-volatile-Read\(_{sb}\)
  \((ys\@[Write_{sb} True a' sop' v' A' L' R' W']) = \{\} \)
  by (auto simp add: outstanding-refs-append suspends\(_j\) [symmetric]
split-suspends\(_j\))

have acq-simp:
  acquired True \( ys \@ \) [Write_{sb} True a' sop' v' A' L' R' W']
  (acquired True \( ?\)take-sb\(_j\) \( O_j \)) =
  acquired True \( ys \) (acquired True \( ?\)take-sb\(_j\) \( O_j \)) \( \cup \) A' \( - \) R'
  by (simp add: acquired-append)

from flush-store-buffer-append [where \( sb=ys\@[Write_{sb} True a' sop' v' A' L' R' W'] \)]
and \( sb'=zs\), simplified,
OF j-bound-ts' is\(_j\) [simplified split-suspends\(_j\)] cph [simplified suspends\(_j\)]
ts'\(_j\) [simplified split-suspends\(_j\)] refl lp [simplified split-suspends\(_j\)] reads-consis-ys
hist-consis' [simplified split-suspends\(_j\)] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends\(_j\)]
no-volatile-Read\(_{sb}\)-volatile-reads-consistent [OF no-vol-read], where
\( S=\)share \( ?\)drop-sb \( S \)

obtain is\(_j'\) \( R_j' \) where

461
is\_j^\prime\); instrs zs @ is\_sbj = is\_j^\prime @ prog-instrs zs and

steps-ys: (?ts\_i, flush ?drop-sb m, share ?drop-sb S) ⇒ d^* 

(?ts\_j[j:=\text{(last-prog}} \ (\text{hd-prog} p) (\text{Write}_s b \ True \ a^\prime \ s^\prime \ v^\prime \ A^\prime \ L^\prime \ R^\prime \ W^\prime \# zs)) \ (ys@[\text{Write}_s b \ True \ a^\prime \ s^\prime \ v^\prime \ A^\prime \ L^\prime \ R^\prime \ W^\prime])

\text{(is \ (-,-,-) ⇒ d^* (?ts-ys,?m-ys,?shared-ys))}

by (auto simp add: acquired-append outstanding-refs-append)

from i-bound have i-bound-ys: i < length ?ts-ys
by auto

from i-bound\_neq-i-j
have ts-ys-i: ?ts-ysli = (p\_sb, i\_sb, ?s\_sb, ()),
D\_sb, acquired True sb \ O\_sb, release sb (dom S\_sb) \ R\_sb
by simp
note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is\_sb]
have a ∈ read-only (share (ys@[\text{Write}_s b True \ a^\prime \ s^\prime \ v^\prime \ A^\prime \ L^\prime \ R^\prime \ W^\prime]) (share ?drop-sb S))
apply cases
apply (auto simp add: Let-def is\_sb)
done

with a-A^\prime
show False
by (simp add: share-append in-read-only-convs)

next
assume ∃ A L R W ys zs. suspends\_j = ys @ Ghost\_sb A L R W # zs ∧ a ∈ A
then
obtain A^\prime L^\prime R^\prime W^\prime ys zs where
split-suspends\_j: suspends\_j = ys @ Ghost\_sb A^\prime L^\prime R^\prime W^\prime# zs
(is suspends\_j = ?suspends) and
a-A^\prime: a ∈ A^\prime
by blast

from valid-program-history [OF j-bound jth]
have causal-program-history is\_sbj sb\_j.
then have cph: causal-program-history is\_sbj ?suspends
apply –
apply (rule causal-program-history-suffix \text{[where sb=?take-sb\_j]})
apply (simp only: split-suspends\_j [symmetric] suspends\_j)

462
apply (simp add: split-suspends)
done

from ts \_j neq-i-j j-bound
have ts'\_j: ?ts'\_j = (hd-prog p \_j \_suspends\_j, is\_j,
  \_v\_sbj \_j \_dom \_v\_sbj = read-tmps \_suspends\_j),()
  \_D\_j, acquired True \_take\_sb\_j \_O\_j, release \_take\_sb\_j (dom \_S\_sb) \_R\_j)
  by auto
from valid-last-prog [OF j-bound jth] have last-prog: last-prog p \_j sb\_j = p\_j.
then have hp: last-prog p \_j \_suspends\_j = p\_j
  apply
  apply (rule last-prog-same-append [where sb=?take-sb])
  apply (simp only: split-suspends [symmetric] \_suspends\_j)
  apply simp
  done

from valid-reads [OF j-bound jth]
have reads-consis-j: reads-consistent False \_O\_j m\_sb sb\_j.
from reads-consistent-flush-all-until-volatile-write [OF \_valid-ownership-and-sharing
  \_S\_sb \_ts\_sb' j-bound
  jth reads-consis-j]
have reads-consis-m-j: reads-consistent True (acquired True \_take\_sb\_j \_O\_j) m\_suspends\_j
  by (simp add: m \_suspends\_j)
from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound neq-i-j ts\_sb'-i
  jth]
have outstanding-refs is-Write\_sb \_drop\_sb \_outstanding-refs is-non-volatile-Read\_sb
  \_suspends\_j = {}
  by (simp add: \_suspends\_j)
from reads-consistent-flush-independent [OF this reads-consis-m-j]
have reads-consis-flush-suspend: reads-consistent True (acquired True \_take\_sb\_j \_O\_j)
  (flush \_drop\_sb m) \_suspends\_j.

hence reads-consis-ys: reads-consistent True (acquired True \_take\_sb\_j \_O\_j)
  (flush \_drop\_sb m) (ys@[\_Ghost\_sb A' L' R' W'])
  by (simp add: split-suspends \_suspends\_j reads-consistent-append)
from valid-write-sops [OF j-bound jth]
have \_\forall \_sop\_write-sops (?\_take\_sb\_j@?\_suspends). \_valid-sop \_sop
  by (simp add: split-suspends \_suspends\_j [symmetric] \_suspends\_j)
then obtain valid-sops-take: \_\forall \_sop\_write-sops ?\_take\_sb\_j. \_valid-sop \_sop and
valid-sops-drop: \_\forall \_sop\_write-sops (ys@[\_Ghost\_sb A' L' R' W']). \_valid-sop \_sop
  apply (simp only: write-sops-append)
  apply auto
  done

from read-tmps-distinct [OF j-bound jth]
have distinct-read-tmps (?\_take\_sb\_j@\_suspends\_j)
  by (simp add: split-suspends \_suspends\_j [symmetric] \_suspends\_j)

463
then obtain
read-tmps-take-drop: read-tmps ?take-sb \( j \cap \) read-tmps suspends \( j \) = \{\} and
distinct-read-tmps-drop: distinct-read-tmps suspends \( j \)
apply (simp only: split-suspends \( j \) [symmetric] suspends \( j \))
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound jth]
have h-consis:
  history-consistent \( \vartheta_{sbj} (hd-prog p_j (?take-sb_j @ suspends_j)) \) (?take-sb_j @ suspends_j)
apply (simp only: split-suspends \( j \) [symmetric] suspends \( j \))
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
proof –
  from last-prog have last-prog p_j (?take-sb_j @ ?drop-sb_j) = p_j
  by simp
  from last-prog-hd-prog-append' [OF h-consis] this
  have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
  by (simp only: split-suspends_j [symmetric] suspends_j)
  moreover
  have last-prog (hd-prog p_j (?take-sb_j @ suspends_j)) ?take-sb_j =
  last-prog (hd-prog p_j suspends_j) ?take-sb_j
  apply (simp only: split-suspends_j [symmetric] suspends_j)
  by (rule last-prog-hd-prog-append)
  ultimately show ?thesis
  by (simp add: split-suspends_j [symmetric] suspends_j)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
h-consis] last-prog-hd-prog
have hist-consis': history-consistent \( \vartheta_{sbj} (hd-prog p_j suspends_j) \) suspends_j
  by (simp add: split-suspends_j [symmetric] suspends_j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-j]
have no-vol-read: outstanding-refs is-volatile-Read_sb
  (ys@[Ghost sb A' L' R' W']) = \{\}
  by (auto simp add: outstanding-refs-append suspends_j [symmetric]
split-suspends_j )

have acq-simp:
  acquired True (ys @ [Ghost sb A' L' R' W'])
  (acquired True ?take-sb_j O_j) =
  acquired True ys (acquired True ?take-sb_j O_j) \cup A' – R'
  by (simp add: acquired-append)

from flush-store-buffer-append [where sb=ys@[Ghost sb A' L' R' W'] and sb'=zs, simplified,
OF j-bound-ts' is_j [simplified split-suspends_j] cph [simplified suspends_j]
ts′ \cdot j \text{ [simplified split-suspends]} \text{ refl lp [simplified split-suspends]} \text{ reads-consis-ys}
hist-consis′ \text{ [simplified split-suspends]} \text{ valid-sops-drop}
distinct-read-tmps-drop \text{ [simplified split-suspends]}
no-volatile-Read sb-volatile-reads-consistent \text{ [OF no-vol-read]}, \text{ where}
\mathcal{S}=\text{share} \ ?\text{drop-sb} \ \mathcal{S}]

\textbf{obtain} \ \text{i-bound′ where}
is_{j}′: \text{instrs} \ zs @ is_{sbj} = is_{j}′ @ \text{prog-instrs} \ zs \ \text{and}
steps-ys: (\?ts′, \text{flush} \ ?\text{drop-sb} \ m, \ \text{share} \ ?\text{drop-sb} \ \mathcal{S}) \ \Rightarrow \ d^* \n
(?ts′[j]:=\text{last-prog}
(\text{hd-prog} \ p_{j} (\text{Ghost} \ A′ \ L′ \ R′ \ W′ \# \ z)) \ (\text{ys}@[\text{Ghost} \ A′ \ L′ \ R′ \ W′]),
is_{j}′,
\mathcal{D}_{j} \ \Rightarrow \ \text{read-tmps} \ zs),
(\).

\mathcal{D}_{j} ∨ \text{outstanding-refs} \ \text{is-volatile-Write}_{sb} \ (\text{ys}@[\text{Ghost} \ A′ \ L′ \ R′ \ W′]) \neq \{\}, acquired \ \text{True} \ \text{ys} \ (\text{acquired} \ \text{True} \ ?\text{take-sb} \ O_{j} \cup A′ − \mathcal{R}_{j}′],
\text{flush} \ (\text{ys}@[\text{Ghost} \ A′ \ L′ \ R′ \ W′]) \ (\text{flush} \ ?\text{drop-sb} \ m),
\text{share} \ (\text{ys}@[\text{Ghost} \ A′ \ L′ \ R′ \ W′] \ (\text{share} \ ?\text{drop-sb} \ S))
(is \ (-,\cdot,\cdot) \Rightarrow \ d^* \ (\text{?ts-ys},?m-ys,?shared-ys))
by \ (\text{auto simp add: acquired-append})

\textbf{from} \ \text{i-bound′ have} \ \text{i-bound-ys:} \ i < \text{length} \ \text{?ts-ys}
by \ \text{auto}

\textbf{from} \ \text{i-bound′ neq-i-j}
\textbf{have} \ \text{ts-ys-i:} \ \text{?ts-ysli} = (p_{sb}, \text{is}_{sb},\emptyset_{sb},()),
\mathcal{D}_{sb}, \ \text{acquired} \ \text{True} \ \text{sb} \ \mathcal{O}_{sb}, \ \text{release} \ \text{sb} \ (\text{dom} \ \mathcal{S}_{sb}) \ \mathcal{R}_{sb})
by \ \text{simp}
\textbf{note} \ \text{conflict-computation} = \text{rtranclp-trans} \ [\text{OF steps-flush-sb steps-ys}]

\textbf{from} \ \text{safe-reach-safe-rtrancl} \ [\text{OF safe-reach conflict-computation}]
\textbf{have} \ \text{safe-delayed} \ (\text{?ts-ys},?m-ys,?shared-ys).

\textbf{from} \ \text{safe-delayedE} \ [\text{OF this} \ \text{i-bound-ys} \ \text{ts-ys-i,} \ \text{simplified} \ \text{is}_{sb}] \ \text{non-vol a-not-acq}
\textbf{have} \ a \in \ \text{read-only} \ (\text{share} \ (\text{ys}@[\text{Ghost} \ A′ \ L′ \ R′ \ W′]) \ (\text{share} \ ?\text{drop-sb} \ \mathcal{S})),
apply \ \text{cases}
apply \ \text{(auto simp add: Let-def is}_{sb})
done

\textbf{with} \ a-A′
\textbf{show} \ \text{False}
by \ \text{(simp add: share-append in-read-only-convs)}
\textbf{qed}
\textbf{qed}

\textbf{note} \ \text{non-volatile-unowned-others} = \text{this}

\{\text{assume} \ \text{a-in: a} \in \ \text{read-only} \ \text{(share} \ \text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ \mathcal{S}.)\text{assume} \ \text{nv:} \ \neg \text{volatile}
\textbf{have} \ a \in \ \text{read-only} \ \text{(share} \ \text{sb} \ \mathcal{S}_{sb})\}

465
proof (cases a ∈ O sb ∪ all-acquired sb)
case True
from share-all-until-volatile-write-thread-local’ [OF ownership-distinct-ts sb
sharing-consis-ts sb i-bound ts sb-i True] True a-in
show ?thesis
  by (simp add: S read-only-def)
next
case False
from read-only-share-unowned [OF weak-consis-drop - a-in] False
  acquired-all-acquired [of True ?take-sb O sb]
  all-acquired-append [of ?take-sb ?drop-sb]
  have a-ro-shared: a ∈ read-only S
    by auto
  have a /∈ ∪ (λ(-, - , - , sb , - , -). all-shared (takeWhile (Not ⌷ is-volatile-Write sb) sb)) ' set ts sb)
proof –
  { fix k p k is k d k O k R k
    assume k-bound: k < length ts sb
    assume ts-k: ts sb ! k = (p k, is k, d k, O k, R k)
    assume a-in: a ∈ all-shared (takeWhile (Not ⌷ is-volatile-Write sb) sb)
    have False
      proof (cases k=i)
        case True
          with False ts sb-i ts-k a-in
          all-shared-acquired-or-owned [OF sharing-consis [OF k-bound ts-k]]
          all-shared-append [of takeWhile (Not ⌷ is-volatile-Write sb) sb]
          dropWhile (Not ⌷ is-volatile-Write sb) sb] show False by auto
      next
        case False
          from rels-nv-cond [rule-format, OF nv k-bound [simplified leq] False [symmetric]
            ]
          ts-sim [rule-format, OF k-bound] ts-k
          have a /∈ dom (release (takeWhile (Not ⌷ is-volatile-Write sb) sb) (dom (S sb))
            )
          R k)
            by (auto simp add: Let-def)
          with dom-release-takeWhile [of sb (dom (S sb)) R k]
          obtain
            a-rels: a /∈ dom R k and
            a-shared: a /∈ all-shared (takeWhile (Not ⌷ is-volatile-Write sb) sb)
            by auto
          with False a-in show ?thesis
            by auto
          qed
        }
      thus ?thesis
        by (auto simp add: in-set-conv-nth)
  qed
  with read-only-shared-all-until-volatile-write-subset’ [of ts sb S sb] a-ro-shared
  have a ∈ read-only S sb
    by (auto simp add: S)
from read-only-share-unowned′ [OF weak-consis-sb read-only-unowned [OF i-bound ts\_sb\_i] False this]

show ?thesis .

qed

} note non-vol-ro-reduction = this

have valid-own′: valid-ownership S\_sb′ ts\_sb′

proof (intro-locales)

show outstanding-non-volatile-ref-owned-or-read-only S\_sb′ ts\_sb′

proof (cases volatile)

case False

from outstanding-non-volatile-ref-owned-or-read-only [OF i-bound ts\_sb\_i]

have non-volatile-owned-or-read-only False S\_sb O\_sb sb.

then

have non-volatile-owned-or-read-only False S\_sb O\_sb (sb@[Read\_sb False a t (m\_sb a)])

using access-cond′ False non-vol-ro-reduction

by (auto simp add: non-volatile-owned-or-read-only-append)

from outstanding-non-volatile-ref-owned-or-read-only-nth-update [OF i-bound this]

show ?thesis by (auto simp add: False ts\_sb′ sb′ O\_sb′ S\_sb′)

next

case True

from outstanding-non-volatile-ref-owned-or-read-only [OF i-bound ts\_sb\_i]

have non-volatile-owned-or-read-only False S\_sb O\_sb sb.

then

have non-volatile-owned-or-read-only False S\_sb O\_sb (sb@[Read\_sb True a t (m\_sb a)])

using True

by (simp add: non-volatile-owned-or-read-only-append)

from outstanding-non-volatile-ref-owned-or-read-only-nth-update [OF i-bound this]

show ?thesis by (auto simp add: True ts\_sb′ sb′ O\_sb′ S\_sb′)

qed

next

show outstanding-volatile-writes-unowned-by-others ts\_sb′

proof –

have out: outstanding-refs is-volatile-Write\_sb (sb@[Read\_sb volatile a t (m\_sb a)]) ⊆ outstanding-refs is-volatile-Write\_sb sb

by (auto simp add: outstanding-refs-append)

have all-acquired (sb@[Read\_sb volatile a t (m\_sb a)]) ⊆ all-acquired sb

by (auto simp add: all-acquired-append)

from outstanding-volatile-writes-unowned-by-others-store-buffer [OF i-bound ts\_sb\_i out this]

show ?thesis by (simp add: ts\_sb′ sb′ O\_sb′)

qed

next

show read-only-reads-unowned ts\_sb′

proof (cases volatile)

case True

have r: read-only-reads (acquired True (takeWhile (Not ◦ is-volatile-Write\_sb))
\[ \text{(sb @ [Read}_{sb} \text{ volatile a t (m}_{sb} a)]) O_{sb} \]
\[ \text{(dropWhile (Not \circ \text{is-volatile-Write}_{sb}) (sb @ [Read}_{sb} \text{ volatile a t (m}_{sb} a)])} \subseteq \text{read-only-reads (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \]
\[ O_{sb} \]
\[ \text{(dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \]
\text{apply} (\text{case-tac outstanding-refs (is-volatile-Write}_{sb}) sb = \{\})
\text{apply} (\text{simp-all add: outstanding-vol-write-take-drop-append}
\text{ acquired-append read-only-reads-append True})
\text{done}

\text{have} O_{sb} \cup \text{all-acquired (sb @ [Read}_{sb} \text{ volatile a t (m}_{sb} a)])} \subseteq O_{sb} \cup \text{all-acquired sb}
\text{by} (\text{simp add: all-acquired-append})

\text{from} \text{read-only-reads-unowned-nth-update [OF i-bound ts}_{sb} \text{-i r this]}
\text{show} ?\text{thesis}
\text{by} (\text{simp add: ts}_{sb} \text{'} O_{sb} \text{'} sb')
\text{next}
\text{case False}
\text{show} ?\text{thesis}
\text{proof} (\text{unfold-locales})
\text{fix} n \ m
\text{fix} p_n \ i s_n \ \mathcal{O}_n \ \mathcal{R}_n \ \mathcal{D}_n \ \mathcal{O}_m \ \mathcal{R}_m \ \mathcal{D}_m \ \mathcal{O}_m \ \mathcal{R}_m \ \mathcal{D}_m \ \mathcal{O}_m \ \mathcal{R}_m
\text{assume} \ n\text{-bound: } n < \text{length ts}_{sb} \text{'}
\text{and} \ m\text{-bound: } m < \text{length ts}_{sb} \text{'}
\text{and} \ neq-n-m: n \neq m
\text{and} \ nth: ts_{sb} \text{'!n} = (p_n, i s_n, \mathcal{D}_n, \mathcal{O}_m, \mathcal{R}_n)
\text{and} \ mth: ts_{sb} \text{'!m} = (p_m, i s_m, \mathcal{D}_m, \mathcal{O}_m, \mathcal{R}_m)
\text{from} \ n\text{-bound} \text{have} n\text{-bound': } n < \text{length ts}_{sb} \text{ by} (\text{simp add: ts}_{sb} \text{'})
\text{from} \ m\text{-bound} \text{have} m\text{-bound': } m < \text{length ts}_{sb} \text{ by} (\text{simp add: ts}_{sb} \text{'})
\text{have} \text{acq-eq: } (O_{sb} \text{'} \cup \text{all-acquired sb'}) = (O_{sb} \cup \text{all-acquired sb})
\text{by} (\text{simp add: all-acquired-append sb'} O_{sb} \text{'})
\text{show} (O_m \cup \text{all-acquired sb_m}) \cap
\text{read-only-reads (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n)} O_n)
\text{(dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n)} = \{\}
\text{proof} (\text{cases } m=\text{i})
\text{case True}
\text{with} \text{neq-n-m} \text{have} neq-n-i: n \neq i
\text{by} \text{auto}
\text{with} \ n\text{-bound nth i-bound} \text{have} nth': ts_{sb} \text{'!n} = (p_n, i s_n, \mathcal{D}_n, O_n, R_n)
\text{by} (\text{auto simp add: ts}_{sb} \text{'})
\text{note} \text{read-only-reads-unowned [OF } n\text{-bound'} \text{i-bound} \text{ neq-n-i nth'} ts_{sb} \text{-i]}
\text{moreover}
\text{note} \text{acq-eq}
\text{ultimately show} ?\text{thesis}
\text{using} True ts_{sb} \text{-i nth m-bound'} m\text{-bound'}
by (simp add: ts_{sb}'

next

case False

note neq-m-i = this

with m-bound mth i-bound have mth': ts_{sb}'!m = (p_m, \theta_m, s_{sb}^m, D_m, O_m, R_m)

by (auto simp add: ts_{sb}')

show ?thesis

proof (cases n=i)


case True

note read-only-reads-unowned [OF i-bound m-bound' neq-m-i [symmetric] ts_{sb}-i mth']

moreover

note acq-eq

moreover

note non-volatile-unowned-others [OF m-bound' neq-m-i [symmetric] mth']

ultimately show ?thesis

using True ts_{sb}-i nth m-bound' m-bound' neq-m-i

apply (case-tac outstanding-refs (is-volatile-Write_{sb}) sb = {})

apply (clarsimp simp add: outstanding-vol-write-take-drop-append

acquired-append read-only-reads-append ts_{sb}' sb'O_{sb}')

done

next

case False

with n-bound nth i-bound have nth': ts_{sb}'!n = (p_n, \theta_n, s_{sb}^n, D_n, O_n, R_n)

by (auto simp add: ts_{sb}')

from read-only-reads-unowned [OF n-bound' m-bound' neq-n-m nth'mth'] False neq-m-i

show ?thesis

by (clarsimp)

qed

qed

show ownership-distinct ts_{sb}'

proof

have all-acquired (sb @ [Read_{sb} volatile a t (m_{sb} a)]) \subseteq all-acquired sb

by (auto simp add: all-acquired-append)

from ownership-distinct-instructions-read-value-store-buffer-independent [OF i-bound ts_{sb}-i this]

show ?thesis by (simp add: ts_{sb}' sb'O_{sb}')

qed

qed

have valid-hist': valid-history program-step ts_{sb}'

proof

from valid-history [OF i-bound ts_{sb}-i]

have hicons: history-consistent \theta_{sb} (hd-prog p_{sb} sb) sb.

from load-tmps-read-tmps-distinct [OF i-bound ts_{sb}-i]

have t-notin-reads: t \notin read-tmps sb

by (auto simp add: is_{sb})

from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
have t-notin-writes: t $\notin \bigcup (\text{fst} \ ' \ \text{write-sops} \ \text{sb} \ )$
by (auto simp add: is sb)

from valid-write-sops [OF i-bound ts sb-i]
have valid-sops: \( \forall \text{sop} \in \text{write-sops} \ \text{sb}. \ \text{valid-sop} \ \text{sop} \)
by auto
from load-tmps-fresh [OF i-bound ts sb-i]
have t-fresh: t $\notin \ \text{dom} \ \partial \ \text{sb}$
using is sb
by simp

from valid-implies-valid-prog-hd [OF i-bound ts sb-i valid]
have history-consistent (\( \partial \ \text{sb}(t \rightarrow t) \ a))
(\text{hd-prog} \ p \ \text{sb}@ [\text{Read} \ \text{sb volatile} \ a \ t (\text{m sb} \ a)])
(\text{sb}@ [\text{Read} \ \text{sb volatile} \ a \ t (\text{m sb} \ a)])
using t-notin-writes valid-sops t-fresh hcons
apply –
apply (rule history-consistent-appendI)
apply (auto simp add: hd-prog-append-Read sb)
done

from valid-history-nth-update [OF i-bound this]
show ?thesis
by (auto simp add: ts sb' sb' O sb' \partial sb')
qed

from reads-consistent-unbuffered-snoc [OF buf-None refl valid-reads [OF i-bound ts sb-i]
volatile-cond ]
have reads-consis': \( \text{reads-consistent} \ \text{False} \ \text{O sb} \ \text{m sb} \ \text{(sb} @ [\text{Read} \ \text{sb volatile} \ a \ t (\text{m sb} \ a)]) \)
by (simp split: if-split-asm)

from valid-reads-nth-update [OF i-bound this]
have valid-reads': valid-reads \text{m sb} \ \text{ts sb}' \by (simp add: ts sb' sb' O sb')

have valid-sharing': valid-sharing \text{S sb}' \text{ts sb}'
proof (intro-locales)
from outstanding-non-volatile-writes-unshared [OF i-bound ts sb-i]
have non-volatile-writes-unshared \text{S sb} \ (\text{sb} @ [\text{Read} \ \text{sb volatile} \ a \ t (\text{m sb} \ a)])
by (auto simp add: non-volatile-writes-unshared-append)
from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared \text{S sb}' \text{ts sb}'
by (simp add: ts sb' sb' S sb')
next
from sharing-consis [OF i-bound ts sb-i]
have sharing-consistent \text{S sb} \ \text{O sb} \ \text{sb}.
then
have sharing-consistent \text{S sb} \ \text{O sb} \ (\text{sb} @ [\text{Read} \ \text{sb volatile} \ a \ t (\text{m sb} \ a)])
by (simp add: sharing-consistent-append)
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' \) \( O_{sb}' \) \( sb' \) \( S_{sb}' \))

next

note read-only-unowned [OF i-bound \( ts_{sb}-i \)]
from read-only-unowned-nth-update [OF i-bound this]
show read-only-unowned \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( S_{sb}' \) \( ts_{sb}' \) \( sb' \) \( O_{sb}' \))

next

from unowned-shared-nth-update [OF i-bound this]
show unowned-shared \( S_{sb}' \) \( ts_{sb}' \) by (simp add: \( ts_{sb}' \) \( O_{sb}' \) \( S_{sb}' \))

next

from no-outstanding-write-to-read-only-memory [OF i-bound \( ts_{sb}-i \)]
have no-write-to-read-only-memory \( S_{sb} \).
hence no-write-to-read-only-memory \( S_{sb} \) (\( sb@\) [Read_{sb} volatile a t (m_{sb} a)])
  by (simp add: no-write-to-read-only-memory-append)
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
show no-outstanding-write-to-read-only-memory \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' \) \( S_{sb}' \) \( sb' \))

qed

have tmps-distinct': tmps-distinct \( ts_{sb}' \)
proof (intro-locales)
from load-tmps-distinct [OF i-bound \( ts_{sb}-i \)]
have distinct-load-tmps \( is_{sb}' \)
  by (auto split: instr.splits simp add: \( is_{sb}' \))
from load-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-distinct \( ts_{sb}' \) by (simp add: \( ts_{sb}' \))

next

from read-tmps-distinct [OF i-bound \( ts_{sb}-i \)]
have distinct-read-tmps \( sb \).
moreover
from load-tmps-read-tmps-distinct [OF i-bound \( ts_{sb}-i \)]
have \( t \notin \) read-tmps \( sb \)
  by (auto simp add: \( is_{sb}' \))
ultimately have distinct-read-tmps (\( sb@\) [Read_{sb} volatile a t (m_{sb} a)])
  by (auto simp add: distinct-read-tmps-append)
from read-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct \( ts_{sb}' \) by (simp add: \( ts_{sb}' \) \( sb' \))

next

from load-tmps-read-tmps-distinct [OF i-bound \( ts_{sb}-i \)]
  load-tmps-distinct [OF i-bound \( ts_{sb}-i \)]
have load-tmps \( is_{sb}' \) \( \cap \) read-tmps (\( sb@\) [Read_{sb} volatile a t (m_{sb} a)]) = {}
  by (clarsimp simp add: read-tmps-append \( is_{sb}' \))
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct \( ts_{sb}' \) by (simp add: \( ts_{sb}' \) \( sb' \))

qed

have valid-sops': valid-sops \( ts_{sb}' \)
proof
from valid-store-sops [OF i-bound \( ts_{sb}-i \)]
have valid-store-sops′ : ∀ sop∈store-sops is_{sb}'.
    valid-sop sop
by (auto simp add: is_{sb})
from valid-write-sops [OF i-bound ts_{sb}-i]
have valid-write-sops′ : ∀ sop∈write-sops (sb@ [Read_{sb} volatile a t (m_{sb} a)]).
    valid-sop sop
by (auto simp add: write-sops-append)
from valid-sops-nth-update [OF i-bound valid-write-sops valid-store-sops']
show ?thesis by (simp add: ts_{sb}' sb')
qed

have valid-dd′ : valid-data-dependency ts_{sb}'
proof
from data-dependency-consistent-instrs [OF i-bound ts_{sb}-i]
have dd-is: data-dependency-consistent-instrs (dom θ_{sb}') is_{sb}'
    by (auto simp add: is_{sb} θ_{sb} θ_{sb}')
from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' ∩ ∪ (fst ' write-sops (sb@ [Read_{sb} volatile a t (m_{sb} a)])) = {}
    by (auto simp add: write-sops-append is_{sb})
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show ?thesis by (simp add: ts_{sb}' sb')
qed

have load-tmps-fresh′ : load-tmps-fresh ts_{sb}'
proof
from load-tmps-fresh [OF i-bound ts_{sb}-i]
have load-tmps (Read volatile a t # is_{sb}') ∩ dom θ_{sb} = {}
    by (auto simp add: is_{sb})
moreover
from load-tmps-distinct [OF i-bound ts_{sb}-i] have t ∉ load-tmps is_{sb}'
    by (auto simp add: is_{sb})
ultimately have load-tmps is_{sb}' ∩ dom (θ_{sb}(t ↦→ (m_{sb} a))) = {}
    by auto
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}' sb' θ_{sb}')
qed

have enough-flushs′ : enough-flushs ts_{sb}'
proof
from clean-no-outstanding-volatile-Write_{sb} [OF i-bound ts_{sb}-i]
have ¬ D_{sb} →→ outstanding.refs is-volatile-Write_{sb} (sb@[Read_{sb} volatile a t (m_{sb} a)]) = {}
    by (auto simp add: outstanding.refs-append)
from enough-flushs-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}' sb' D_{sb}')
qed

have valid-program-history′ : valid-program-history ts_{sb}'
proof
from valid-program-history [OF i-bound ts_{sb}-i]
have causal-program-history is_{sb} sb .
then have causal’: causal-program-history is_{sb}’ (sb@[Read_{sb} volatile a t (m_{sb} a)])
  by (auto simp: causal-program-history-Read is_{sb})
from valid-last-prog [OF i-bound ts_{sb} i]
have last-prog p_{sb} sb = p_{sb}.
hence last-prog p_{sb} (sb@[Read_{sb} volatile a t (m_{sb} a)]) = p_{sb}
  by (simp add: last-prog-append-Read_{sb})
from valid-program-history-nth-update [OF i-bound causal’ this]
show ?thesis
  by (simp add: ts_{sb}’ sb’)
qued

show ?thesis
proof (cases outstanding-refs is-volatile-Write_{sb} sb = {})
case True
from True have flush-all: takeWhile (Not ◦ is-volatile-Write_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
from True have suspend-nothing: dropWhile (Not ◦ is-volatile-Write_{sb}) sb = []
  by (auto simp add: outstanding-refs-conv)
hence suspends-empty: suspends = []
  by (simp add: suspends)
from suspends-empty is-sim have is: is = Read volatile a t # is_{sb}’
  by (simp add: is_{sb})
with suspends-empty ts-i
have ts-i: ts!i = (p_{sb}, Read volatile a t # is_{sb}’, \varnothing_{sb},() ,
  D, acquired True ?take-sb \mathcal{O}_{sb}, release ?take-sb (dom \mathcal{S}_{sb}) \mathcal{R}_{sb})
  by simp
from direct-memop-step. Read
have (Read volatile a t # is_{sb}’, \varnothing_{sb}, () , m ,
  D, acquired True ?take-sb \mathcal{O}_{sb}, release ?take-sb (dom \mathcal{S}_{sb}) \mathcal{R}_{sb}, S) \rightarrow
  (is_{sb}’, \varnothing_{sb}(t \mapsto m a), () , m , D, acquired True ?take-sb \mathcal{O}_{sb},
  release ?take-sb (dom \mathcal{S}_{sb}) \mathcal{R}_{sb}, S).
from direct-computation.concurrent-step.Memop [OF i-bound’ ts-i this]
have (ts, m, S) \Rightarrow (ts !i := (p_{sb}, is_{sb}’, \varnothing_{sb}(t \mapsto m a), () ,
  D, acquired True ?take-sb \mathcal{O}_{sb}, release ?take-sb (dom \mathcal{S}_{sb}) \mathcal{R}_{sb})), m, S).
moreover
from flush-all-until-volatile-write-Read-commute [OF i-bound ts_{sb} i [simplified is_{sb} ]]
have flush-commute: flush-all-until-volatile-write
  (ts_{sb}!i := (p_{sb},is_{sb}’, \varnothing_{sb}(t \mapsto m_{sb} a), sb@[Read_{sb} volatile a t (m_{sb} a)], D_{sb}, \mathcal{O}_{sb}, \mathcal{R}_{sb})))
  m_{sb} =
  flush-all-until-volatile-write ts_{sb} m_{sb}.
have flush-all-until-volatile-write ts_{sb} m_{sb} a = m_{sb} a
proof

have \( \forall j < \text{length } ts_{sb}. i \neq j \rightarrow \)
    \( (\text{let } (\cdot, \cdot, sb_{j}, \cdot, \cdot, \cdot) = ts_{sb}[j] \)
        \( \text{in } a \notin \text{outstanding-refs } \text{is-non-volatile-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j})) \)

proof

let \(?take-sb_{j} = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}) \)
let \(?drop-sb_{j} = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}) \)
assume \( a\in: a \in \text{outstanding-refs } \text{is-non-volatile-Write}_{sb} ?take-sb_{j} \)
with \( \text{outstanding-refs-takeWhile } \text{where } P' = \text{Not } \circ \text{is-volatile-Write}_{sb} \)
have \( a\notin: a \notin \text{outstanding-refs } \text{is-non-volatile-Write}_{sb} \)

by auto

with \( \text{non-volatile-owned-or-read-only-outstanding-non-volatile-writes} \)
[OF \( \text{outstanding-non-volatile-refs-owned-or-read-only } [\text{OF } j\text{-bound } jth] \)]

have \( j\text{-owns: } a \in O_{j} \cup \text{all-acquired } sb_{j} \)
by auto

with \( \text{ownership-distinct } [\text{OF } i\text{-bound } j\text{-bound } \text{neq-i-j } ts_{sb}\cdot i jth] \)

have \( a\text{-not-owns: } a \notin O_{sb} \cup \text{all-acquired } sb \)
by blast

from \( \text{non-volatile-owned-or-read-only-append } [\text{of False } S_{sb} O_{j} ?take-sb_{j} ?drop-sb_{j}] \)
\( \text{outstanding-non-volatile-refs-owned-or-read-only } [\text{OF } j\text{-bound } jth] \)

have \( \text{non-volatile-owned-or-read-only } False S_{sb} O_{j} ?take-sb_{j} \)
by simp

from \( \text{non-volatile-owned-or-read-only-outstanding-non-volatile-writes } [\text{OF this}] a\text{-in} \)

have \( j\text{-owns-drop: } a \in O_{j} \cup \text{all-acquired } ?take-sb_{j} \)
by auto

from \( \text{rels-cond } [\text{rule-format, OF } j\text{-bound } [\text{simplified leq} \text{ neq-i-j}] ts\cdot \sim] \)
[rule-format, OF \( j\text{-bound}\) jth]

have \( \text{no-unsharing-release } ?take-sb_{j} (\text{dom } (S_{sb})) R_{j} a \neq \text{Some False} \)
by (auto simp add: Let-def)

\{
assume \( a \in \text{acquired } True sb O_{sb} \)
with \( \text{acquired-all-acquired-in } [\text{OF this}] \text{ownership-distinct } [\text{OF i-bound } j\text{-bound } \text{neq-i-j } ts_{sb}\cdot i jth] \)
\( j\text{-owns} \)

have \( \text{False} \)
by auto
\}

moreover

\{

474
assume a-ro: a ∈ read-only (share ?drop-sb S)
from read-only-share-unowned-in [OF weak-consis-drop a-ro] a-not-owns
acquired-all-acquired [of True ?take-sb O_sb]
all-acquired-append [of ?take-sb ?drop-sb]
have a ∈ read-only S
  by auto
with share-all-until-volatile-write-thread-local [OF ownership-distinct-ts_sb
sharing-consis-ts_sb j-bound jth j-owns]
  have a ∈ read-only (share ?take-sb j S_sb)
    by (auto simp add: read-only-def S)
  hence a-dom: a ∈ dom (share ?take-sb j S_sb)
    by (auto simp add: read-only-def domIff)
from outstanding-non-volatile-writes-unshared [OF j-bound jth]
non-volatile-writes-unshared-append [of S_sb ?take-sb ?drop-sb]
have nvw: non-volatile-writes-unshared S_sb ?take-sb ?drop-sb
  by auto
from release-not-unshared-no-write-take [OF this no-unsharing a-dom] a-in
have False by auto
}
moreover
{
  assume a-share: volatile ∧ a ∈ dom (share ?drop-sb S)
  from outstanding-non-volatile-writes-unshared [OF j-bound jth]
  have non-volatile-writes-unshared S_sb sb_j.
  with non-volatile-writes-unshared-append [of S_sb (takeWhile (Not is-volatile-Write sb_j) sb_j)
    (dropWhile (Not o is-volatile-Write sb_j) sb_j)]
    have unshared-take: non-volatile-writes-unshared S_sb (takeWhile (Not o is-volatile-Write sb_j) sb_j)
      by clarsimp
  from valid-own have own-dist: ownership-distinct ts_sb
    by (simp add: valid-ownership-def)
  from valid-sharing have sharing-consis S_sb ts_sb
    by (simp add: valid-sharing-def)
  from sharing-consistent-share-all-until-volatile-write [OF own-dist this i-bound ts_sb-i]
  have sc: sharing-consistent S (acquired True ?take-sb O_sb) ?drop-sb
    by (simp add: S)
  from sharing-consistent-share-all-shared
  have dom (share ?drop-sb S) ⊆ dom S ∪ all-shared ?drop-sb
    by auto
  also from sharing-consistent-all-shared [OF sc]
  have ... ⊆ dom S ∪ acquired True ?take-sb O_sb by auto
  also from acquired-all-acquired all-acquired-takeWhile
  have ... ⊆ dom S ∪ (O_sb ∪ all-acquired sb) by force
  finally
  have a-shared: a ∈ dom S
    using a-share a-not-owns
    by auto
with share-all-until-volatile-write-thread-local [OF ownership-distinct-ts\sb
sharing-consis-ts\sb j-bound jth j-owns]

have a-dom: a ∈ dom (share ?take-sb\sb \sb S\sb)
by (auto simp add: S domIff)
from release-not-unshared-no-write-take [OF unshared-take no-unsharing
a-dom] a-in
have False by auto

ultimately show False
using access-cond'
by auto
qed

thus ?thesis
by (fastforce simp add: Let-def)
qed

from flush-all-until-volatile-write-buffered-val-conv
[OF True i-bound ts\sb-i this]
show ?thesis
by (simp add: buf-None)
qed

hence m-a: m a = m\sb a
by (simp add: m)

have tmps-commute: \sb\sb\sb \sb t \mapsto (m\sb a)
(\sb\sb\sb \sb |' (dom \sb\sb\sb - \{t\})) \sb t \mapsto (m\sb a))
apply (rule ext)
apply (auto simp add: restrict-map-def domIff)
done

from suspend-nothing
have suspend-nothing': (dropWhile (Not \circ is-volatile-Write\sb sb') sb') = []
by (simp add: sb')

from D
have D': D\sb = (D \lor outstanding-refs is-volatile-Write\sb (sb@[Read\sb volatile a t (m\sb a)]) ≠ \{\})
by (auto simp: outstanding-refs-append)

have (ts\sb',m\sb SB\sb') ~ (ts[i := (p\sb SB\sb', i\sb SB\sb')(t\mapsto m a),()], D, acquired True ?take-sb
O\sb SB\sb,release ?take-sb (dom S\sb SB\sb) R\sb SB\sb]), m,S)
apply (rule sim-config.intros)
apply (simp add: m flush-commute ts\sb'\sb O\sb SB\sb' R\sb SB\sb' \sb \sb SB\sb' D\sb' )
using share-all-until-volatile-write-Read-commute [OF i-bound ts\sb-i [simplified is\sb SB\sb]]
apply (simp add: S SB\sb' SB\sb' sb' O\sb SB\sb' R\sb SB\sb' \sb SB\sb')
using leq
apply (simp add: ts\sb')
using i-bound i-bound' ts-sim ts-i True D'}

476
apply (clarsimp simp add: Let-def nth-list-update
outstanding-refs-conv m-a ts sb' O sb' R sb' S sb' D sb' suspend-nothing'
flush-all acquired-append release-append
split: if-split-asn )
apply (rule tmps-commute)
done

ultimately show \( \text{thesis} \)
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct'
valid-sops' valid-dd' load-tmps-fresh' enough-flushs'
valid-program-history valid'
m sb' S sb' by (auto simp del: fun-upd-apply )
next
case False
then obtain r where r-in: r \( \in \) set sb and volatile-r: is-volatile-Write sb r
by (auto simp add: outstanding-refs-conv)
from takeWhile-dropWhile-real-prefix
[OF r-in, of (Not o is-volatile-Write sb), simplified, OF volatile-r]
obtain a' v' sb'' sop' A' L' R' W' where
sb-split: sb = takeWhile (Not o is-volatile-Write sb) sb @ Write sb True a' sop' v' A' L'
R' W' # sb''
and
drop: dropWhile (Not o is-volatile-Write sb) sb = Write sb True a' sop' v' A' L' R' W' #
sb''
apply (auto)
subgoal for y ys
apply (case-tac y)
apply auto
done
done
from drop suspends have suspends: suspends = Write sb True a' sop' v' A' L' R' W' #
sb''
by simp

have (ts, m, S) \( \Rightarrow_d^* \) (ts, m, S) by auto

moreover

note flush-commute = flush-all-until-volatile-write-Read-commute [OF i-bound ts sb-i
[simplified is sb] ]

have Write sb True a' sop' v' A' L' R' W' \( \in \) set sb
by (subst sb-split) auto

from dropWhile-append1 [OF this, of (Not o is-volatile-Write sb)]
have drop-app-comm:
(dropWhile (Not o is-volatile-Write sb) (sb @ [Read sb volatile a t (m sb a)])) =
dropWhile (Not ◦ is-volatile-Write_{sb}) sb @ [Read_{sb} volatile a t (m_{sb} a)]

by simp

from load-tmps-fresh [OF i-bound ts_{sb}-i]
have t ∉ dom 0_{sb}
  by (auto simp add: is_{sb})
then have tmps-commute:
  0_{sb} " (dom 0_{sb} − read-tmps sb") =
  0_{sb} " (dom 0_{sb} − insert t (read-tmps sb")

apply −
apply (rule ext)
apply auto
done

from D
have D': D_{sb} = (D ∨ outstanding-refs is-volatile-Write_{sb} (sb@[Read_{sb} volatile a t (m_{sb} a)]) ≠ { })
  by (auto simp: outstanding-refs-append)

have (ts_{sb}',m_{sb},S_{sb}) ∼ (ts,m,S)
  apply (rule sim-config.intros)
  apply (simp add: m flush-commute ts_{sb}' O_{sb}' R_{sb}' θ_{sb}')
  using share-all-until-volatile-write-Read-commute [OF i-bound ts_{sb}-i [simplified is_{sb}]]
  apply (simp add: S S_{sb}' ts_{sb}' sb'O_{sb}' R_{sb}' θ_{sb}')
  using leq
  apply (simp add: ts_{sb}'
  using i-bound i-bound' ts-sim ts-i is-sim D'
  apply (clarsimp simp add: Let-def nth-list-update is-sim drop-app-comm
    read-tmps-append suspends prog-instrs-append-Read_{sb} instrs-append-Read_{sb}
    hd-prog-append-Read_{sb}
    drop is_{sb} ts_{sb}' sb'O_{sb}' R_{sb}' θ_{sb}' D_{sb}' acquired-append takeWhile-append1 [OF r-in]
    volatile-r split: if-split-asm)
  apply (simp add: drop tmps-commute)+
  done

ultimately show ?thesis
  using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-dd'
    valid-sops' load-tmps-fresh' enough-flushs'
    valid-program-history' valid'
    m_{sb}' S_{sb}'
  by (auto simp del: fun-upd-apply )
qed

next
  case (SBHWriteNonVolatile a D f A L R W)
  then obtain
    is_{sb}: is_{sb} = Write False a (D, f) A L R W# is_{sb}' and
    O_{sb}': O_{sb} = O_{sb} and
    R_{sb}': R_{sb} = R_{sb} and
    0_{sb}': 0_{sb}' = 0_{sb} and
    D_{sb}': D_{sb}' = D_{sb} and
\[sb' = \text{sb}@[\text{Write}_{sb} \text{ False a } (D, f) (f \cdot \text{sb}) A L R W] \text{ and} \]
\[m_{sb'} = m_{sb} \text{ and} \]
\[S_{sb'} = S_{sb} \]
by auto

from data-dependency-consistent-instrs [OF i-bound ts_{sb-i}]
have D-tmps: D \subseteq \text{dom } \emptyset_{sb}
by (simp add: is_{sb})

from safe-memop-flush-sb [simplified is_{sb}]
obtain a-owned': a \in \text{acquired True sb } \mathcal{O}_{sb} \text{ and} a\text{-unshared': a } \notin \text{dom (share } ?\text{drop-sb } S) \text{ and}
rels-cond: \forall j < \text{length ts. } i \neq j \longrightarrow a \notin \text{dom (released (ts!j))}
by cases auto

from a-owned' acquired-all-acquired
have a-owned'': a \in \mathcal{O}_{sb} \cup \text{all-acquired sb}
by auto

\{  
fix j
fix p_j is_j \mathcal{O}_j \mathcal{R}_j D_j \emptyset_j sb_j
assume j-bound: j < \text{length ts}_{sb}
assume ts_{sb-j}: ts_{sb}!j = (p_j,is_j,\emptyset_j,\mathcal{O}_j,\mathcal{R}_j)
assume neq-i-j: i \neq j
have a \notin \mathcal{O}_j \cup \text{all-acquired sb}_j
proof  
from ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb-i} ts_{sb-j}] a\text{-owned''}
show ?thesis
by auto
qed
\} note a\text{-unowned-others = this}

have a\text{-unshared: a } \notin \text{dom (share sb } S_{sb})
proof
assume a-share: a \in \text{dom (share sb } S_{sb})
from valid-sharing have sharing-consis S_{sb} ts_{sb}
by (simp add: valid-sharing-def)
from in-shared-sb-share-all-until-volatile-write [OF this i-bound ts_{sb-i} a\text{-owned'' a-share}]
have a \in \text{dom (share } ?\text{drop-sb } S)
by (simp add: S)
with a\text{-unshared'}
show False
by auto
qed

479
have valid-own': valid-ownership $S_{sb'}$ $ts_{sb'}$
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only $S_{sb'}$ $ts_{sb'}$
proof
from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound $ts_{sb}$]
have non-volatile-owned-or-read-only False $S_{sb}$ $O_{sb}$ $sb$.

with a-owned'
have non-volatile-owned-or-read-only False $S_{sb}$ $O_{sb}$ ($sb$ @ [Write$sb$ False $a$ (D,$f$) (f $θ_{sb}$) A L R W])
  by (simp add: non-volatile-owned-or-read-only-append)
from outstanding-non-volatile-refs-owned-or-read-only-only-nth-update [OF i-bound this]
show ?thesis by (simp add: $ts_{sb}$' $is_{sb}$ $sb'$ $O_{sb}$' $S_{sb}$')
qed

next
show outstanding-volatile-writes-unowned-by-others $ts_{sb'}$
proof
  have outstanding-refs is-volatile-Write$sb$ ($sb$ @ [Write$sb$ False a (D,$f$) (f $θ_{sb}$) A L R W])
    ⊆ outstanding-refs is-volatile-Write$sb$ $sb$
    by (auto simp add: outstanding-refs-append)
  from outstanding-volatile-writes-unowned-by-others-store-buffer [OF i-bound $ts_{sb}$-i this]
  show ?thesis by (simp add: $ts_{sb}$' $is_{sb}$ $sb'$ $O_{sb}$' all-acquired-append)
qed

next
show read-only-reads-unowned $ts_{sb'}$
proof
  have r: read-only-reads (acquired True (takeWhile (Not ◦ is-volatile-Write$sb$) (sb @ [Write$sb$ False a (D,$f$) (f $θ_{sb}$) A L R W])) $O_{sb}$)
    (dropWhile (Not ◦ is-volatile-Write$sb$) (sb @ [Write$sb$ False a (D,$f$) (f $θ_{sb}$) A L R W]))
    ⊆ read-only-reads (acquired True (takeWhile (Not ◦ is-volatile-Write$sb$) sb) $O_{sb}$)
    (dropWhile (Not ◦ is-volatile-Write$sb$) sb)
    apply (case-tac outstanding-refs (is-volatile-Write$sb$) sb = { })
    apply (simp-all add: outstanding-vol-write-take-drop-appends acquired-append-read-only-reads-append)
  done
  have $O_{sb}$ ∪ all-acquired ($sb$ @ [Write$sb$ False a (D,$f$) (f $θ_{sb}$) A L R W]) ⊆ $O_{sb}$ ∪ all-acquired $sb$
    by (simp add: all-acquired-append)

from read-only-reads-unowned-nth-update [OF i-bound $ts_{sb}$-i r this]
show ?thesis
  by (simp add: $ts_{sb}$' $O_{sb}$' $sb'$)
qed
next
show ownership-distinct ts_{sb}'
proof -

from ownership-distinct-instructions-read-value-store-buffer-independent [OF i-bound ts_{sb}-i]
show ?thesis by (simp add: ts_{sb}' is_{sb} sb' O_{sb}' all-acquired-append)
qed

have valid-hist': valid-history program-step ts_{sb}'
proof -

from valid-history [OF i-bound ts_{sb}-i]
have history-consistent θ_{sb} (hd-prog p_{sb} sb) sb.
with valid-write-sops [OF i-bound ts_{sb}-i] D-tmps
valid-implies-valid-prog-hd [OF i-bound ts_{sb}-i valid]
have history-consistent θ_{sb} (hd-prog p_{sb} (sb@[Write_{sb} False a (D,f) (f θ_{sb}) A L R W]))
apply -
apply (rule history-consistent-appendII)
apply (auto simp add: hd-prog-append-Write)
done

from valid-history-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}' is_{sb} sb' O_{sb}' θ_{sb}')
qed

have valid-reads': valid-reads m_{sb} ts_{sb}'
proof -

from valid-reads [OF i-bound ts_{sb}-i]
have reads-consistent False O_{sb} m_{sb} sb.
from reads-consistent-snoc-Write_{sb} [OF this]
have reads-consistent False O_{sb} m_{sb} (sb @ [Write_{sb} False a (D,f) (f θ_{sb}) A L R W]).
from valid-reads-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}' is_{sb} sb' O_{sb}' θ_{sb}')
qed

have valid-sharing': valid-sharing S_{sb}' ts_{sb}'
proof (intro-locales)

from outstanding-non-volatile-writes-unshared [OF i-bound ts_{sb}-i] a-unshared
have non-volatile-writes-unshared S_{sb}
(sb @ [Write_{sb} False a (D,f) (f θ_{sb}) A L R W])
by (auto simp add: non-volatile-writes-unshared-append)
from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared S_{sb}' ts_{sb}'
by (simp add: ts_{sb}' is_{sb} sb' O_{sb}' θ_{sb}' S_{sb}')
next

from sharing-consis [OF i-bound ts_{sb}-i]
have sharing-consistent S_{sb} O_{sb} sb.
then
have sharing-consistent S_{sb} O_{sb} (sb @ [Write_{sb} False a (D,f) (f θ_{sb}) A L R W])
by (simp add: sharing-consistent-append)
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' O_{sb}' sb' S_{sb}' \))
next
from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts_{sb}-i]]
show read-only-unowned \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( S_{sb}' ts_{sb}' O_{sb}' \))
next
from unowned-shared-nth-update [OF i-bound ts_{sb}-i subset-refl]
show unowned-shared \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' is_{sb} sb' O_{sb}' \theta_{sb}' S_{sb}' \))
next
from a-unshared
have a \( \not\in \) read-only (share \( sb S_{sb} \))
  by (auto simp add: read-only-def dom-def)
with no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb}-i]

have no-write-to-read-only-memory \( S_{sb} \) \( sb @ [Write_{sb} False a (D,f) (f \theta_{sb}) A L R W] \)
  by (simp add: no-write-to-read-only-memory-append)
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
show no-outstanding-write-to-read-only-memory \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( S_{sb}' ts_{sb}' sb' \))
qed

have tmps-distinct \( ' \) : tmps-distinct \( ts_{sb}' \)
proof (intro-locales)
from load-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-load-tmps \( is_{sb}' \)
  by (auto split: instr.splits simp add: \( is_{sb} \))
from load-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-distinct \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' is_{sb} sb' O_{sb}' \theta_{sb}' \))
next
from read-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-read-tmps \( sb. \)
  hence distinct-read-tmps \( sb @ [Write_{sb} False a (D,f) (f \theta_{sb}) A L R W] \)
  by (simp add: distinct-read-tmps-append)
from read-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' is_{sb} sb' O_{sb}' \theta_{sb}' \))
next
from load-tmps-distinct [OF i-bound ts_{sb}-i]
load-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps \( is_{sb}' \cap \) read-tmps \( sb @ [Write_{sb} False a (D,f) (f \theta_{sb}) A L R W] \) = \{ \}
  by (clarsimp simp add: read-tmps-append is_{sb})
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' is_{sb} sb' O_{sb}' \theta_{sb}' \))
have valid-sops' valid-sops ts\_sb' 
proof --
from valid-store-sops [OF i-bound ts\_sb-i]
obtain valid-Df: valid-sop (D,f) and
valid-store-sops' \(\forall \) sop\in store-sops is\_sb', valid-sop
by (auto simp add: is\_sb)
from valid-Df valid-write-sops [OF i-bound ts\_sb-i]
have valid-write-sops' \(\forall \) sop\in write-sops (sb@ [Write\_sb False a (D,f) (f \theta\_sb) A L R W]).
valid-sop sop
by (auto simp add: write-sops-append)
from valid-sops-nth-update [OF i-bound valid-write-sops' valid-store-sops']
show ?thesis
by (simp add: ts\_sb' is\_sb sb' O\_sb' \theta\_sb')
qed

have valid-dd' valid-data-dependency ts\_sb'
proof --
from data-dependency-consistent-instrs [OF i-bound ts\_sb-i]
obtain D-indep: D \(\cap\) load-tmps is\_sb' = {} and
dd-is: data-dependency-consistent-instrs (dom \theta\_sb') is\_sb'
by (auto simp add: is\_sb \theta\_sb')
from load-tmps-write-tmps-distinct [OF i-bound ts\_sb-i] D-indep
have load-tmps is\_sb' \(\cap\)
\(\bigcup\) (fst ' write-sops (sb@ [Write\_sb False a (D,f) (f \theta\_sb) A L R W])) = {}
by (auto simp add: write-sops-append is\_sb)
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show ?thesis
by (simp add: ts\_sb' is\_sb sb' O\_sb' \theta\_sb')
qed

have load-tmps-fresh' load-tmps-fresh ts\_sb'
proof --
from load-tmps-fresh [OF i-bound ts\_sb-i]
have load-tmps is\_sb' \(\cap\) dom \theta\_sb = {}
by (auto simp add: is\_sb)
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis
by (simp add: ts\_sb' is\_sb sb' O\_sb' \theta\_sb')
qed

have enough-flushs' enough-flushs ts\_sb'
proof --
from clean-no-outstanding-volatile-Write\_sb [OF i-bound ts\_sb-i]
have \(\neg\) D\_sb \(\rightarrow\) outstanding-refs is-volatile-Write\_sb (sb@ [Write\_sb False a (D,f) (f \theta\_sb) A L R W]) = {}
by (auto simp add: outstanding-refs-append )
from enough-flushs-nth-update [OF i-bound this]
show ?thesis

483
by (simp add: ts\sb\' \cdot D\sb\' )
  qed

  \textbf{have} \text{ valid-program-history}\': \text{ valid-program-history} ts\sb\'
  \textbf{proof} ~
  from \text{ valid-program-history} \ [OF \text{i-bound} ts\sb\'-i]
  \textbf{have} \text{ causal-program-history} is\sb\' sb .
  then \textbf{have} causal': \text{ causal-program-history} is\sb\' \ (sb\@[Write sb False a (D,f) (f \vartheta sb) \ A L R W])
  \textbf{by} (auto simp: causal-program-history-Write is\sb)
  from \text{ valid-last-prog} \ [OF \text{i-bound} ts\sb\'-i]
  \textbf{have} \text{ last-prog} p\sb\ sb = p\sb .
  \textbf{hence} \text{ last-prog} p\sb\ (sb\@[Write sb False a (D,f) (f \vartheta sb) \ A L R W]) = p\sb
  \textbf{by} (simp add: last-prog-append-Write sb)
  from \text{ valid-program-history- nth-update} \ [OF \text{i-bound causal'} \text{ this}]
  \textbf{show} \ ?thesis
  \textbf{by} (simp add: ts\sb\' \cdot sb\' )
  qed

  from \text{ valid-store-sops} \ [OF \text{i-bound} ts\sb\'-i, \text{ rule-format}]
  \textbf{have} \text{ valid-sop} (D,f) \textbf{by} (auto simp add: is\sb)
  then \textbf{interpret} \text{ valid-sop} (D,f) .
  \textbf{show} \ ?thesis
  \textbf{proof} (cases outstanding-refs is-volatile-Write sb = \{\})
  \textbf{case} True
  from True \textbf{have} flush-all: \text{ takeWhile} (Not \circ is-volatile-Write sb) sb = sb
  \textbf{by} (auto simp add: outstanding-refs-conv)
  from True \textbf{have} suspend-nothing: \text{ dropWhile} (Not \circ is-volatile-Write sb) sb = []
  \textbf{by} (auto simp add: outstanding-refs-conv)
  \textbf{hence} suspends-empty: suspends = []
  \textbf{by} (simp add: suspends)
  from suspends-empty is-sim \textbf{have} is: \text{ is} = \text{ Write False} a (D,f) \ A L R W# is\sb\'
  \textbf{by} (simp add: is\sb)
  with suspends-empty ts-i
  \textbf{have} ts-i: ts\#i = (p\sb , \text{ Write False} a (D,f) \ A L R W# is\sb\' ,
  \vartheta sb (),
  D , acquired True ?take-sb O sb, release ?take-sb (dom (S sb)) R sb )
  \textbf{by} simp
  from direct-memop-step. WriteNonVolatile \ [OF ]
  \textbf{have} \text{ (Write False} a (D, f) \ A L R W# is\sb\' ,
  \vartheta sb , () , m , D , acquired True ?take-sb O sb , release ?take-sb (dom (S sb)) R sb , S ) \rightarrow
  (is\sb\' ,
∀j<\text{length } ts_{sb}. \ i \neq j \rightarrow
\begin{align*}
\text{let } (-\cdot,\cdot,-,sb_{j},\cdot\cdot\cdot) &= ts_{sb} ! j \\
\text{in a} \notin \text{outstanding-refs is-non-volatile-Write}_{sb_{j}} \ (\text{takeWhile } (\text{Not} \circ \text{is-volatile-Write}_{sb_{j}})\ sb_{j})
\end{align*}

proof
{
\begin{align*}
\text{fix } j \ p_{j} \ i_{j} \ O_{j} \ R_{j} \ D_{j} &\ \text{acq}_{j} \ x_{j} \ sb_{j} \\
\text{assume } j\text{-bound: } j < \text{length } ts_{sb} \\
\text{assume } neq\text{-i-j: } i \neq j \\
\text{assume } j\text{th: } ts_{sb} ! j = (p_{j},is_{j},xs_{j},sb_{j},D_{j},O_{j},R_{j}) \\
\text{have a} \notin \text{outstanding-refs is-non-volatile-Write}_{sb_{j}} \ (\text{takeWhile } (\text{Not} \circ \text{is-volatile-Write}_{sb_{j}})\ sb_{j}) \\
\end{align*}

proof
{
\begin{align*}
\text{assume a-in: } a \in \text{outstanding-refs is-non-volatile-Write}_{sb_{j}} \ (\text{takeWhile } (\text{Not} \circ \text{is-volatile-Write}_{sb_{j}})\ sb_{j}) \\
\text{hence a} \in \text{outstanding-refs is-non-volatile-Write}_{sb_{j}}\ sb_{j} \\
\text{using outstanding-refs-append } [\text{of is-non-volatile-Write}_{sb_{j}} \ (\text{takeWhile } (\text{Not} \circ \text{is-volatile-Write}_{sb_{j}})\ sb_{j})] \\
\text{(dropWhile } (\text{Not} \circ \text{is-volatile-Write}_{sb_{j}})\ sb_{j})] \\
\text{by auto} \\
\text{with non-volatile-owned-or-read-only-outstanding-non-volatile-writes} \\
[\text{OF outstanding-non-volatile-refs-owned-or-read-only } [\text{OF } j\text{-bound } j\text{th}]] \\
\text{have j\text{-owns: } a \in O_{j} \cup \text{all-acquired } sb_{j}} \\
\text{by auto} \\
\text{from j\text{-owns a\text{-owned” ownership-distinct } [\text{OF } i\text{-bound } j\text{-bound } neq\text{-i-j } ts_{sb}\text{-i } j\text{th}]} \\
\text{show False} \\
\text{by auto} \\
\text{qed} \\
\end{align*}
\}

\text{thus } \text{?thesis by } (\text{fastforce simp add: Let-def}) \\
\text{qed}

\text{note flush-commute = flush-all-until-volatile-write-append-non-volatile-write-commute} \\
[\text{OF } \text{True } i\text{-bound } ts_{sb}\text{-i } \text{this}]

\text{from suspend-nothing} \\
\text{have suspend-nothing': } (\text{dropWhile } (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb') = [] \\
\text{by (simp add: sb')}
from $D$

have $D' : D' \triangleq (D \lor \text{outstanding-refs is-volatile-Write}_{sb}
(sb \in \{\text{Write}_{sb} \text{ False} a (D, f) (f \in D)\} A L R W) \neq \{\})$
by (auto simp: outstanding-refs-append)

have $(ts_{sb}', m_{sb}, S_{sb}') \sim
(ts[i := (p_{sb}, is_{sb}', \vartheta_{sb}(.)), D, acquired True \text{ take-sb } O_{sb},
release ?\text{take-sb } (\text{dom } (S_{sb})) R_{sb}]),
\text{sub}(\vartheta_{sb}), S)
apply (rule sim-config.intros)
apply (simp add: m flush-commute ts_{sb}' O_{sb}' R_{sb}' sb'
\vartheta_{sb}' D_{sb}')
using share-all-until-volatile-write-Write-commute
[OF i-bound ts_{sb}-i [simplified is_{sb}]]
apply (clarsimp simp add: \text{leq} i-bound i-bound'
\text{ts-sim ts-i True} D_{sb}')
apply (clarsimp simp add: \text{Le-def nth-list-update}
\text{outstanding-refs-conv ts_{sb}' O_{sb}' R_{sb}' S_{sb}' \vartheta_{sb}' sb'
\vartheta_{sb}' D_{sb}' \text{suspend-nothing} \text{ flush-all acquired-append release-append split: if-split-asm})
done

ultimately
show ?thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmpls-distinct' valid-sops'
valid-dd' load-tmps-fresh' enough-flushs'
valid-program-history' valid' m_{sb}' S_{sb}'
by (auto simp del: fun-upd-apply)

next

case False

then obtain r where r-in: $r \in \text{set } sb$ and volatile-r: is-volatile-Write_{sb} $r$
by (auto simp add: outstanding-refs-conv)
from takeWhile-dropWhile-real-prefix
[OF r-in, of (Not \circ is-volatile-Write_{sb}), simplified, OF volatile-r]
obtain $a' \text{ v'} sb''/sop' A' L' R' W'$ where
sb-split: $sb = \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb \circ \text{Write}_{sb} \text{ True } a' \text{ sop' v'} A' L'$
R' W'# sb''
and
\text{drop: } dropWhile (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb = \text{Write}_{sb} \text{ True } a' \text{ sop' v'} A' L' R' W'# sb''
apply (auto)
\text{subgoal for } y \text{ ys}
apply (case-tac y)
apply auto
done
done

done

486
from drop suspends have suspends: suspends = Write\sb True a' sop' v' A' L' R' W'\# sb''
  by simp

have (ts, m, S) ⇒ d*(ts, m, S) by auto

moreover

note flush-commute =
  flush-all-until-volatile-write-append-unflushed [OF False i-bound ts\sb-i]

have Write\sb True a' sop' v' A' L' R' W' ∈ set sb
  by (subst sb-split) auto

note drop-app = dropWhile-append1 [OF this, simplified]

from D have D': D\sb' = (D ∨ outstanding-refs is-volatile-Write\sb (sb@[Write \sb False a (D, f) (f θ \sb)]) A L R W) ≠ {})
  by (auto simp: outstanding-refs-append)

have (ts\sb', m\sb, S\sb') ~ (ts, m, S)
  apply (rule sim-config.intros)
  apply (simp add: m flush-commute ts\sb' O\sb' R\sb' θ\sb' sb')
  using share-all-until-volatile-write-Write-commute
  [OF i-bound ts\sb-i [simplified is\sb]]
  apply (clarsimp simp add: S\sb' ts\sb' sb' O\sb' R\sb' θ\sb' sb)
  using leq
  apply (clarsimp simp add: nth-list-update is-sim drop-app
  read-tmps-append suspends
  prog-instrs-append-Write\sb instrs-append-Write\sb hd-prog-append-Write\sb
  drop is\sb' ts\sb' sb' O\sb' R\sb' S\sb' θ\sb' sb
  \sb' D' \sb' acquired-append takeWhile-append1 [OF r-in]
  volatile-r
  split: if-split-asm)
  done

ultimately show ?thesis
  using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-dd'
  valid-sops' load-tmps-fresh' enough-flushes'
  valid-program-history' valid' m\sb' S\sb'
  by (auto simp del: fun-upd-apply )
  qed

next
case (SBHWriteVolatile a D f A L R W)
then obtain
is\sb': is\sb = Write True a (D, f) A L R W# is\sb' and
O\sb': O\sb' = O\sb and
R\sb': R\sb' = R\sb and

487
\( \theta' = \theta \) and 
\( D' = \text{True} \) and 
\( \text{sb'=sb@[Write}_{sb} \text{ True a (D, f) (f } \theta_{sb} \text{) A L R W]} \) and 
\( m' = m \) and 
\( S' = S \) by auto

```
from data-dependency-consistent-instrs [OF i-bound ts_{sb-i}]
have D-subset: \( D \subseteq \text{dom } \theta_{sb} \)
by (simp add: is_{sb})

from safe-memop-flush-sb [simplified is_{sb}] obtain
a-unowned-others-ts:
\( \forall j < \text{length} (\text{map owned ts}). i \neq j \rightarrow (a \notin (\text{owned}\ (ts!j) \cup \text{dom}\ (\text{released}\ (ts!j)))) \)

and
L-subset: \( L \subseteq A \) and 
A-shared-owned: \( A \subseteq \text{dom}\ (\text{share } ?\text{drop-sb } S) \cup \text{acquired}\ True\ sb\ O_{sb} \) and 
R-acq: \( R \subseteq \text{acquired}\ True\ sb\ O_{sb} \) and 
A-R: \( A \cap R = \{\} \) and 
A-unowned-by-others-ts: 
\( \forall j < \text{length} (\text{map owned ts}). i \neq j \rightarrow (A \cap (\text{owned}\ (ts!j) \cup \text{dom}\ (\text{released}\ (ts!j))) = \{\}) \)

and
a-not-ro': \( a \notin \text{read-only}\ (\text{share } ?\text{drop-sb } S) \)
by cases auto

from a-unowned-others-ts ts-sim leq
have a-unowned-others:
\( \forall j < \text{length} \ ts_{sb}. i \neq j \rightarrow \)
\( \text{(let } (-,r_r,sb\ j,\ O_j,r) = \ts_{sb}\ ! j \text{ in} \)
\( a \notin ) \text{acquired True (takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb\ j) \ O_j \land \)
\( a \notin \text{all-shared (takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb\ j)) \)
apply (clarsimp simp add: Let-def)
subgoal for j
apply (drule-tac x=j in spec)
apply (auto simp add: dom-release-takeWhile)
done
done

have a-not-ro: \( a \notin \text{read-only}\ (\text{share } sb\ S_{sb}) \)
proof
assume a: \( a \in \text{read-only}\ (\text{share } sb\ S_{sb}) \)
from local.read-only-unowned-axioms have read-only-unowned \( S_{sb} \ ts_{sb} \).

from in-read-only-share-all-until-volatile-write' [OF ownership-distinct-ts_{sb}]
sharing-consis-ts_{sb}
(\text{read-only-unowned } S_{sb} \ ts_{sb} \ i-bound \ ts_{sb-i} \ i-unowned-others a)

have a \in \text{read-only}\ (\text{share } ?\text{drop-sb } S)
by (simp add: S)
with a-not-ro' show False by simp
```
from A-unowned-by-others-ts \( ts \)-sim leq

have A-unowned-by-others:
\[ \forall j < \text{length} \, ts_{sb} \cdot i \neq j \rightarrow (let (\ldots, \cdot, O_j, \ldots) = ts_{sb}!j \in A \cap (\text{acquired True (takeWhile (Not \, \text{is-volatile-Write}_{sb}) \, sb_j)} \, O_j \cup \text{all-shared (takeWhile (Not \, \text{is-volatile-Write}_{sb}) \, sb_j)}) = \{\}) \]

apply (clarsimp simp add: Let-def)

subgoal for \( j \)
apply (drule-tac \( x=j \) in spec)
apply (force simp add: dom-release-takeWhile)
done

done

have a-not-acquired-others:
\[ \forall j < \text{length} \, (\text{map } O_s {\cdot} sb \, ts_{sb}) \cdot i \neq j \rightarrow (let (O_j, sb_j) = (\text{map } O_{sb} ts_{sb})!j \in a \not\in \text{all-acquired } sb_j) \]

proof –

{ fix \( j \) \( O_j \) \( sb_j \)
assume j-bound: \( j < \text{length} \, (\text{map owned } ts_{sb}) \)
assume neq-i-j: \( i \neq j \)
assume ts_{sb}-j: (\text{map } O_{sb} ts_{sb})!j = (O_j, sb_j) \)
assume conflict: \( a < \text{all-acquired } sb_j \)
have False
proof –
from j-bound leq
have j-bound\( ^{\prime} \): \( j < \text{length} \, (\text{map owned } ts) \)
by auto
from j-bound have j-bound\( ^{\prime\prime} \): \( j < \text{length} \, ts_{sb} \)
by auto
from j-bound\( ^{\prime} \) have j-bound\( ^{\prime\prime} \): \( j < \text{length } ts \)
by simp

let \( ?\text{take-sb}_j = \text{(takeWhile (Not \, \text{is-volatile-Write}_{sb}) \, sb_j)} \)
let \( ?\text{drop-sb}_j = \text{(dropWhile (Not \, \text{is-volatile-Write}_{sb}) \, sb_j)} \)

from ts-sim [rule-format, OF j-bound\( ^{\prime\prime} \) ts_{sb}-j j-bound\( ^{\prime\prime} \) ”

obtain \( p_j \) suspends_j \( is_{sbj} \) \( R_j \) \( D_{sbj} \) \( D_j \) \( \emptyset_{sbj} \) is_j where
\( ts_{sbj}: ts_{sb} ! j = (p_j, is_{sbj}, \emptyset_{sbj}, sb_j, D_{sbj}, O_j, R_j) \)
and suspends_j: suspends_j = dropWhile (Not \, \text{is-volatile-Write}_{sb}) \, sb_j \)
and \( is_j: \) instrs suspends_j \( \sigma \) \( is_{sbj} = is_j \) \( \sigma \) \( \text{prog-instrs suspends_j} \)
and \( D_j: D_{sbj} = (D_j \lor \text{outstanding-refs is-volatile-Write}_{sb} \, sb_j \neq \{\}) \)
and \( ts_j: ts!j = (\text{hd-prog } p_j \, \text{suspends_j}, i_{sbj}, \emptyset_{sbj} \mid \emptyset_{sbj} - \text{read-tmps suspends_j}),() \)

\( \emptyset_{sbj} \mid \emptyset_{sbj} - \text{read-tmps suspends_j},() \)

\( D_j, \)

acquired True ?take-sb_j \( O_j, \)
release ?take-sb_j (dom \( S_{sb} \) \( R_j)) \)

apply (cases \( ts_{sb!j} \))
apply (force simp add: Let-def)
done

489
from a-unowned-others [rule-format,OF - neq-i-j] ts_{sb-j} j-bound

obtain a-unacq: a \notin \text{acquired} \ True \ \text{?take-sb}_j \ \text{O}_j \ \text{and} \ a\not\in\text{all-shared}\ \text{?take-sb}_j

by auto

have conflict-drop: a \in \text{all-acquired} \ \text{suspends}_j

proof (rule ccontr)

assume a \notin \text{all-acquired} \ \text{suspends}_j

with all-acquired-append [of ?take-sb_j ?drop-sb_j] conflict

have a \in \text{all-acquired} \ ?take-sb_j

by (auto simp add: suspends_j)

from all-acquired-unshared-acquired [OF this a-not-shared] a-unacq

show False by auto

qed

from j-bound'' i-bound' have j-bound-ts': j < length ?ts'

by simp

from split-all-acquired-in [OF conflict-drop]

show ?thesis

proof

assume \exists \text{sop} a' \text{v} \text{ys} \text{zs} \text{A L R W}.

\text{suspends}_j = \text{ys} @ \text{Write}_{sb} \text{True} \ a' \text{sop} \text{v} \text{A L R W# zs} \land a \in A

then

obtain a' \text{sop}' v' \text{ys} \text{A' L' R' W' where}

split-suspends_j: \text{suspends}_j = \text{ys} @ \text{Write}_{sb} \text{True} a' \text{sop}' v' A' L' R' W'\# zs

(is \text{suspends}_j = ?\text{suspends}) \ \text{and}

a-A': a \in A'

by blast

from sharing-consis [OF j-bound'' ts_{sb-j}]

have sharing-consis-j: sharing-consistent \text{S}_{sb} \text{O}_j \text{sb}_j.

then have A' R': A' \cap R' = \{\}

by (simp add: sharing-consistent-append [of - - ?take-sb_j ?drop-sb_j; simplified]

\text{suspends}_j \text{[symmetric]} \text{split-suspends}_j \text{sharing-consistent-append})

from valid-program-history [OF j-bound'' ts_{sb-j}]

have causal-program-history \text{is}_{sbj} \text{sb}_j.

then have cph: causal-program-history \text{is}_{sbj} \ ?\text{suspends}

apply --

apply (rule causal-program-history-suffix [where sb=?take-sb_j])

apply (simp only: split-suspends_j [symmetric] \text{suspends}_j)

apply (simp add: split-suspends_j)

done

from ts_j neq-i-j j-bound

have ts''_j: ?ts''_j = (\text{hd-prog} p_j \text{suspends}_j, i_j, \text{is}_{sbj} |^j (\text{dom} \text{is}_{sbj} \text{read-tmps} \text{suspends}_j),()
\( D_j \), acquired True ?take-sb \( O_j \), release ?take-sb \( \text{dom } S_{\text{sb}} \) \( R_j \)

by auto

from valid-last-prog \([\text{OF } j\text{-bound}'' t_{\text{sb}}-j]\) have last-prog: last-prog \( p_j \) \( s_{\text{bj}} \) = \( p_j \).
then
have lp: last-prog \( p_j \) suspends\( j \) = \( p_j \)

apply –
apply (rule last-prog-same-append \([\text{where } sb=?take-sb]\))
apply (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
apply simp
done

from valid-reads \([\text{OF } j\text{-bound}'' t_{\text{sb}}-j]\)

have reads-consis-j: reads-consistent False \( O_j \) \( m_{\text{sb}} \) \( s_{\text{bj}} \).

from reads-consistent-flush-all-until-volatile-write \([\text{OF } \text{valid-ownership-and-sharing} \ S_{\text{sb}} t_{\text{sb}}] \)

\( j\text{-bound}'' t_{\text{sb}}-j \) this
have reads-consis-m-j: reads-consistent True (acquired True ?take-sb \( O_j \) \( m \) suspends\( j \))
by (simp add: \( m \) suspends\( j \))

from outstanding-non-write-non-vol-reads-drop-disj \([\text{OF } i\text{-bound } j\text{-bound}'' \text{neq-i-j} t_{\text{sb}}-i t_{\text{sb}}-j]\)

have outstanding-refs is-Write\( s_{\text{b}} \) ?drop-sb \( \cap \) outstanding-refs is-non-volatile-Read\( s_{\text{b}} \) suspends\( j \) = \{\}
by (simp add: suspends\( j \))
from reads-consistent-flush-independent \([\text{OF } \text{this } \text{reads-consis-m-j}]\)

have reads-consis-flush-suspend: reads-consistent True (acquired True ?take-sb \( O_j \)) (flush ?drop-sb \( m \) suspends\( j \)).

hence reads-consis-ys: reads-consistent True (acquired True ?take-sb \( O_j \)) (flush ?drop-sb \( m \) \( (\text{ys@[Write}_{s_{\text{b}}} \text{True } a' \text{sop' v' A' L' R' W']}))
by (simp add: split-suspends\( j \) reads-consistent-append)

from valid-write-sops \([\text{OF } j\text{-bound}'' t_{\text{sb}}-j]\)

have \( \forall \text{sop}\in\text{write-sops } (?\text{take-sb}@?\text{suspends}) \). valid-sop \( \text{sop} \)
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
then obtain valid-sops-take: \( \forall \text{sop}\in\text{write-sops } (?\text{take-sb}). \text{valid-sop } \text{sop} \) and
valid-sops-drop: \( \forall \text{sop}\in\text{write-sops } (\text{ys@[Write}_{s_{\text{b}}} \text{True } a' \text{sop' v' A' L' R' W']}). \text{valid-sop } \text{sop} \)
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct \([\text{OF } j\text{-bound}'' t_{\text{sb}}-j]\)

have distinct-read-tmps (?take-sb@?suspends\( j \))
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
then obtain
read-tmps-take-drop: read-tmps ?take-sb \( j \) \( \cap \) read-tmps suspends\( j \) = \{\}
and
distinct-read-tmps-drop: distinct-read-tmps suspends\( j \)

491
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound” ts^sb-j]
have h-consis:
  history-consistent θ^sb_j (hd-prog p_j (?take-sb_j@suspends_j)) (?take-sb_j@suspends_j)
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
proof
  from last-prog have last-prog p_j (?take-sb_j@?drop-sb_j) = p_j
    by simp
from last-prog-hd-prog-append’ [OF h-consis] this
have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
  by (simp only: split-suspends_j [symmetric] suspends_j)
moreover
have last-prog (hd-prog p_j (?take-sb_j @ suspends_j)) ?take-sb_j =
  last-prog (hd-prog p_j suspends_j) ?take-sb_j
  by (rule last-prog-hd-prog-append)
ultimately show ?thesis
  by (simp add: split-suspends_j [symmetric] suspends_j)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog
  have hist-consis’: history-consistent θ^sb_j (hd-prog p_j suspends_j) suspends_j
  by (simp add: split-suspends_j [symmetric] suspends_j)
  from reads-consistent-drop-volatile-writes-no-volatile-reads
    [OF reads-consis-ys]
  have no-vol-read: outstanding-refs is-volatile-Read^sb
    (ys@[Write^sb True a’ sop’ v’ A’ L’ R’ W’]) = {}
    by (auto simp add: outstanding-refs-append suspends_j [symmetric] split-suspends_j)
  have acq-simp:
    acquired True (ys @ [Write^sb True a’ sop’ v’ A’ L’ R’ W’])
      (acquired True ?take-sb_j O_j) =
    acquired True ys (acquired True ?take-sb_j O_j) ∪ A’ − R’
    by (simp add: acquired-append)
from flush-store-buffer-append [where sb=ys@[Write^sb True a’ sop’ v’ A’ L’ R’ W’]
and sb’=zs, simplified,
  OF j-bound-ts’ i_s [simplified split-suspends_j] cph [simplified suspends_j]
ts’j [simplified split-suspends_j] refl lp [simplified split-suspends_j] reads-consis-ys
 hist-consis’ [simplified split-suspends_j] valid-sops-drop
  distinct-read-tmps-drop [simplified split-suspends_j]
no-volatile-Read$_{ab}$-volatile-reads-consistent [OF no-vol-read], where

\[ \mathcal{S} = \text{share } ?\text{drop-sb } \mathcal{S} \]

obtain \( \text{i}_{sbj} \mathcal{R}_{sbj} \) where

\( \text{i}_{sbj} \): \( \text{instrs } zs \circ \text{is}_{sbj} = \text{i}_{sbj} \circ @ \text{prog-instrs } zs \) and

\( \text{steps-ys} \): (?ts', flush ?drop-sb m, share ?drop-sb \( \mathcal{S} \)) \( \Rightarrow_d^* \)

\( (?\text{ts'})_j = (\text{last-prog} p_j (\text{Write}_{ab} \text{ True } a' \text{ sop}' \text{v'} A' L' R' W' \# zs)) (\text{ys}@[\text{Write}_{ab} \text{ True } a' \text{ sop}' \text{v'} A' L' R' W')] \)

\( \text{i}_{sbj} \)

\( \theta_{sbj} |_1 (\text{dom } \theta_{sbj} - \text{read-tmps } zs) \)

\( (), \text{True}, \text{acquired } \text{True } \text{ys} (\text{acquired } \text{True } ?\text{take-sb } j \mathcal{O}_j) \cup A' - R', \mathcal{R}'_j] \),

flush (\text{ys}@[\text{Write}_{ab} \text{ True } a' \text{ sop}' \text{v'} A' L' R' W']) (flush ?drop-sb m),

share (\text{ys}@[\text{Write}_{ab} \text{ True } a' \text{ sop}' \text{v'} A' L' R' W']) (share ?drop-sb \( \mathcal{S} \))

\( (\text{is } (\cdot, \cdot, \cdot) \Rightarrow_d^* (\text{?ts-ys'}, \text{?m-ys'}, \text{?shared-ys})) \)

by (auto simp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: \( i < \text{length } \text{?ts-ys} \)

by auto

from i-bound' neq-i-j

have ts-ys-i: \( ?\text{ts-ys}!i = (p_{ab}, \text{is}_{sb}, \theta_{sb}, \cdot) \),

\( \mathcal{D}_{sb}, \text{acquired } \text{True } \mathcal{S}_{sb}, \text{release } \mathcal{S} \text{ (dom } \mathcal{S}_{sb} \) \( \mathcal{R}_{sb}) \)

by simp

note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]


from safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is_{sb}]

have a-unowned:

\( \forall j < \text{length } ?\text{ts-ys}. \ i \neq j \rightarrow (\text{let } (\mathcal{O}_j) = \text{map owned } ?\text{ts-ys}!j \text{ in } a \notin \mathcal{O}_j) \)

apply cases

apply (auto simp add: Let-def is_{sb})
done

from a-A' a-unowned [rule-format, of j] neq-i-j j-bound' A' R'

show False

by (auto simp add: Let-def)

next

assume \( \exists A L R W ys zs. \text{suspends}_j = \text{ys } @ \text{Ghost}_{sb} A L R W \# zs \land a \in A \)

then

obtain A' L' R' W' ys zs where

split-suspendsj: \( \text{suspends}_j = \text{ys } @ \text{Ghost}_{sb} A' L' R' W' \# zs \)

(is suspendsj = [suspends]) and

a-A': \( a \in A' \)

by blast

from sharing-consis [OF j-bound'' ts_{sb}-j]

have sharing-consis-j: sharing-consistent \( \mathcal{S}_{sb} \mathcal{O}_j sb_j \).

then have A' R': \( A' \cap R' = \{\} \)

493
by (simp add: sharing-consistent-append [of - - ?take-sb ?drop-sb; simplified]
 suspends_j [symmetric] split-suspends_j sharing-consistent-append)

 from valid-program-history [OF j-bound'' ts_{sb-j}]
 have causal-program-history is_{sb-j} sb_j.
 then have cplh: causal-program-history is_{sb-j} ?suspends

 apply --
 apply (rule causal-program-history-suffix [where sb=?take-sb])
 apply (simp only: split-suspends_j [symmetric] suspends_j)
 apply (simp add: split-suspends_j)
 done

 from ts_j neq-i-j j-bound
 have ts'_j: ?ts'_j = (hd-prog p_j suspends_j, is_j,
 \(\lambda_{sb-j} \mid (dom \lambda_{sb-j} - \text{read-tmps suspends}_j)\),
 \(P_j\), acquired True ?take-sb \(O_j\), release ?take-sb \(\text{dom } S_{sb}\) \(R_j\)
 by auto

 from valid-last-prog [OF j-bound'' ts_{sb-j}]
 have last-prog: last-prog p_j sb_j = p_j.

 apply --
 apply (rule last-prog-same-append [where sb=?take-sb])
 apply (simp only: split-suspends_j [symmetric] suspends_j)
 apply simp
 done

 from valid-reads [OF j-bound'' ts_{sb-j}]
 have reads-consis-j: reads-consistent False \(O_j\) m_{sb} sb_j.
 from reads-consistent-flush-all-until-volatile-write [OF \text{valid-ownership-and-sharing}
 \(S_{sb}\) \(ts_{sb-j}\)]
 have reads-consis-m-j: reads-consistent True (acquired True ?take-sb \(O_j\)) m suspends_j
 by (simp add: m suspends_j)

 from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound'' neq-i-j
 \(ts_{sb-j}\) this]
 have outstanding-refs is-Write_{sb} ?drop-sb \(\cap\) outstanding-refs is-non-volatile-Read_{sb}
 suspends_j = {}.
 by (simp add: suspends_j)

 from reads-consistent-flush-independent [OF this reads-consis-m-j]
 have reads-consis-flush-suspend: reads-consistent True (acquired True ?take-sb \(O_j\))
 (flush ?drop-sb m) suspends_j.

 hence reads-consis-ys: reads-consistent True (acquired True ?take-sb \(O_j\))
 (flush ?drop-sb m) (\(\text{ys}@(\text{Ghost}_{sb} A' L' R' W')\))
 by (simp add: split-suspends_j reads-consistent-append)

 from valid-write-sops [OF j-bound'' ts_{sb-j}]
 have \(\forall \text{sop} \in \text{write-sops} \ (\text{?take-sb} @ ?suspends)\). valid-sop sop
 by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain valid-sops-take: ∀ sop∈write-sops ?take-sb_j. valid-sop sop and
valid-sops-drop: ∀ sop∈write-sops (ys@Ghost_{s}\ A’ L’ R’ W’). valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound" ts_{s\b}\j]
have distinct-read-tmps (?take-sb_j@suspends_j)
by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain
read-tmps-take-drop: read-tmps ?take-sb_j ∩ read-tmps suspends_j = {}
and
distinct-read-tmps-drop: distinct-read-tmps suspends_j
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound" ts_{s\b}\j]
have h-consis:
history-consistent θ_{s\b}\j (hd-prog p_j (?take-sb_j@suspends_j)) (?take-sb_j@suspends_j)
apply (simp only: split-suspends_j [symmetric] suspends_j)
done

have last-prog-hd-prog: last-prog (hd-prog p_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
proof —
from last-prog have last-prog p_j (?take-sb_j@?drop-sb_j) = p_j
by simp
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
by (simp only: split-suspends_j [symmetric] suspends_j)
moreover
have last-prog (hd-prog p_j (?take-sb_j @ suspends_j)) ?take-sb_j =
last-prog (hd-prog p_j suspends_j) ?take-sb_j
apply (simp only: split-suspends_j [symmetric] suspends_j)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends_j [symmetric] suspends_j)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog
have hist-consis': history-consistent θ_{s\b}\j (hd-prog p_j suspends_j) suspends_j
by (simp add: split-suspends_j [symmetric] suspends_j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-j]
have no-vol-read: outstanding-refs is-volatile-Read_{s\b}
(ys@Ghost_{s\b} A’ L’ R’ W’) = {}
by (auto simp add: outstanding-refs-append suspends_j [symmetric] split-suspends_j)
have acq-simp:
acquired True (ys @ [Ghost\_sb A' L' R' W'])
(acquired True ?take-sb\_\_j O\_j) =
acquired True ys (acquired True ?take-sb\_\_j O\_j) ∪ A' - R'
by (simp add: acquired-append)

from flush-store-buffer-append [where sb=ys@[Ghost\_sb A' L' R' W'], sb'=zs, simplified, 
OF j-bound-ts' is\_j [simplified split-suspends\_j] cph [simplified suspends\_j]
distinct-read-tmps-drop [simplified split-suspends\_j]
no-volatile-Read\_sb-volatile-reads-consistent [OF no-vol-read], where
S=share ?drop-sb S]

obtain is\_j' \_R\_j' where
is\_j': instrs zs @ is\_sb\_j = is\_j' @ prog-instrs zs and
steps-ys: (?ts', flush ?drop-sb m, share ?drop-sb S) ⇒ a*
(?ts'\_j=(last-prog
(lad-prog p\_j (Ghost\_sb A' L' R' W'@ zs)) (ys@[Ghost\_sb A' L' R' W'])
is\_j',
θ\_sb\_j황 (dom θ\_sb\_j - read-tmps zs),
()),
\_D\_j ∨ outstanding-refs is-volatile-Write\_sb (ys @[Ghost\_sb A' L' R'
W']) ≠ \{\}, acquired True ys (acquired True ?take-sb\_\_j O\_j) ∪ A' - R'/R\_j'
flush (ys@[Ghost\_sb A' L' R' W']) (flush ?drop-sb m),
share (ys@[Ghost\_sb A' L' R' W']) (share ?drop-sb S))
(is (-,-,-) ⇒ a* (?ts-ys,?m-ys,?shared-ys))
by (auto simp add: acquired-append)

from i-bound' have i-bound-ys: i < length ?ts-ys
by auto

from i-bound' neq-i-j
have ts-ys-i: ?ts-ys!! = (p\_sb, is\_sb, θ\_sb, ()).
\_D\_sb, acquired True sb O_\_sb, release sb (dom S\_sb) R_\_sb)
by simp

note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is\_sb]
have a-unowned:
\forall j < length ?ts-ys. i\neq j → (let (O\_j) = map owned ?ts-ys!j in a \notin O\_j)
apply cases
apply (auto simp add: Let-def is\_sb)
done

from a-A' a-unowned [rule-format, of j neq-i-j j-bound' A'-R'
show False
by (auto simp add: Let-def)

496
thus ?thesis
by (auto simp add: Let-def)
qed

have A-unused-by-others:
∀ j < length (map O-sb ts_{sb}). i ≠ j →
  (let (O_j, sb_j) = map O-sb ts_{sb}! j
   in A ∩ outstanding-refs is-volatile-Write_{sb} sb_j = {})

proof –

  { 
    fix j O_j sb_j 
    assume j-bound: j < length (map owned ts_{sb}) 
    assume neq-i-j: i ≠ j 
    assume ts_{sb}-j: (map O-sb ts_{sb})!j = (O_j, sb_j) 
    assume conflict: A ∩ outstanding-refs is-volatile-Write_{sb} sb_j ≠ {} 

    have False 
    proof –
      from j-bound leq 
      have j-bound': j < length (map owned ts) 
        by auto 
      from j-bound have j-bound'': j < length ts_{sb} 
        by auto 
      from j-bound' have j-bound'''': j < length ts 
        by simp 

      from conflict obtain a' where 
      a'-in: a' ∈ A and 
      a'-in-j: a' ∈ outstanding-refs is-volatile-Write_{sb} sb_j 
        by auto 

      let ?take-sb_j = (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j) 
      let ?drop-sb_j = (dropWhile (Not ◦ is-volatile-Write_{sb}) sb_j) 

      from ts-sim [rule-format, OF j-bound''''] ts_{sb}-j j-bound'' 
      obtain p_j suspends_{sb} D_{sb} D_j R_j ?sb_j is_j where 
      ts_{sb}-j: ts_{sb}! j = (p_j; is_{sb}; sb_j; D_{sb}; O_j; R_j) and 
      suspends_{sb}: suspends_{sb} = ?drop-sb_j and 
      is_j: intrs suspends_{sb} @ is_{sb} = is_j @ instrs suspends_{sb} and 
      D_j: D_{sb} = (D_j ∨ outstanding-refs is-volatile-Write_{sb} sb_j ≠ {} ) and 
      ts_{sb}: ts_{sb} = (hd-prog p_j suspends_{sb}, is_j, 
      ?sb_j |' (dom ?sb_j = read-tmps suspends_{sb}), (), D_j, 
      acquired True ?take-sb_j O_j, 
      release ?take-sb_j (dom S_{sb}) R_j) 
      apply (cases ts_{sb}[j]) 
      apply (force simp add: Let-def) 
      done 
    }
have \( a' \in \text{outstanding-refs} \text{ is-volatile-Write}_{sb} \) suspends

proof

- from \( a' \in \text{j-bound} \)
  have \( a' \in \text{outstanding-refs} \text{ is-volatile-Write}_{sb} \) (?take-sb \( j \) @ ?drop-sb \( j \))

by simp

thus \( \text{thesis} \)

apply (simp only: outstanding-refs-append suspends)

apply (auto simp add: outstanding-refs-conv dest: set-takeWhileD)

done

qed

from split-volatile-Write_{sb}-in-outstanding-refs [OF this]

obtain \( \text{sop v ys zs A}' \ (L)' \ (R)' \ (W)' \) where

split-suspends

: suspends \( j \) = \( ys \) @ Write_{sb} True \( \text{a}' \in \text{outstanding-refs} \) suspends \( j \) = \( \text{is suspends} \)

by blast

from direct-memop-step.WriteVolatile [where \( \theta\sb = \theta\sb \text{ and } m=\text{flush ?drop-sb m} \)]

have \( \text{Write True a (D, f) A L R # is}_{sb}' \),

\( \theta_{sb}, () \), \( \text{flush ?drop-sb m}, D_{sb}, \text{acquired True sb } O_{sb}, \)

release \( \text{sb (dom } S_{sb}) \) \( R_{sb}, \)

\( \text{share ?drop-sb } S \to \)

\( (is_{sb}', \theta_{sb}, (), (\text{flush ?drop-sb m})(a := f \theta_{sb}), \text{True, acquired True sb } O_{sb} \cup \)

\( \text{A - R, Map.empty,} \)

\( \text{share ?drop-sb } S \oplus W R \ominus A L ) \).

from direct-computation.concurrent-step.Memop [OF i-bound-t's [simplified is_{sb}] ts'-i [simplified is_{sb}] this [simplified is_{sb}]]

have store-step: \( (?ts', \text{flush ?drop-sb m, share ?drop-sb } S ) \Rightarrow_d \)

\( (?ts'[i := (p_{sb}, is_{sb}', \theta_{sb}, (), \text{True, acquired True sb } O_{sb} \cup \text{A - R, Map.empty}), (\text{flush ?drop-sb m})(a := f \theta_{sb}), \text{share ?drop-sb } S \oplus W R \ominus A L ) \}

\( (\text{is - } \Rightarrow_d (?ts\sb-A, ?m-A, ?share-A)) \)

by (simp add: is_{sb})

from i-bound' have i-bound'': \( i < \text{length } ?ts\sb-A \)

by simp

from valid-program-history [OF j-bound'' ts_{sb}-j]

have causal-program-history is_{sbj} sb_{j}.

then have cph: causal-program-history is_{sbj} ?suspends

apply –

apply (rule causal-program-history-suffix [where sb=?take-sb_{j}]

apply (simp only: split-suspends_{j} [symmetric] suspends_{j})

apply (simp add: split-suspends_{j})

done

from ts_{j} neq-i-j j-bound

498
have ts-A-j: ?ts-A!j = (hd-prog p_j (ys @ Write_{sb} True a′ sop v A’ L’ R’ W’# zs), is_{j},
υ_{sbj} l | (dom υ_{sbj} − read-thms (ys @ Write_{sb} True a′ sop v A’ L’ R’ W’# zs)), (), D_j,
acquired True ?take-sb_{j} O_j,release ?take-sb_{j} (dom S_{sb}) R_j)
by (simp add: split-suspends_{j})

from j-bound'' i-bound' neq-i-j have j-bound'''': j < length ?ts-A
by simp

from valid-last-prog [OF j-bound'' ts_{sb}-j] have last-prog: last-prog p_j sb_j = p_j.
then have lp: last-prog p_j ?suspends = p_j
apply −
apply (rule last-prog-same-append [where sb=?take-sb_{j}])
apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
apply simp
done

from valid-reads [OF j-bound'' ts_{sb}-j]
have reads-consis: reads-consistent False O_{j} m_{sb} sb_{j}.

from reads-consistent-flush-all-until-volatile-write [OF (valid-ownership-and-sharing S_{sb} ts_{sb}-j reads-consis)]
have reads-consis-m: reads-consistent True (acquired True ?take-sb_{j} O_{j}) m suspends_{j}
by (simp add: m suspends_{j})

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound'' neq-i-j ts_{sb}-i ts_{sb}-j]
have outstanding-refs is-Write_{sb} ?drop-sb ∩ outstanding-refs is-non-volatile-Read_{sb}
suspends_{j} = {}
by (simp add: suspends_{j})
from reads-consistent-flush-independent [OF this reads-consis-m]
have reads-consis-m: reads-consistent True (acquired True ?take-sb_{j} O_{j})
     (flush ?drop-sb m) suspension_{j}.

from a-unowned-others [rule-format, OF - neq-i-j] j-bound ts_{sb}-j
obtain a-notin-owns-j: a /∈ acquired True ?take-sb_{j} O_{j} and a-unshared: a /∈ all-shared
     ?take-sb_{j}
by auto
from a-not-acquired-others [rule-format, OF - neq-i-j] j-bound ts_{sb}-j
have a-not-acquired-j: a /∈ all-acquired sb_{j}
by auto

from outstanding-non-volatile-refs-owned-or-read-only [OF j-bound'' ts_{sb}-j]
have nvo-j: non-volatile-owned-or-read-only False S_{sb} O_{j} sb_{j}.

have a-no-non-vol-read: a /∈ outstanding-refs is-non-volatile-Read_{sb} ?drop-sb_{j}
proof
assume \( a \in \text{outstanding-refs is-non-volatile-Read}_{sb} \) ?drop-sb\(_j\).

from reads-consistent-drop [OF reads-consis]
have \( \text{rc}: \) reads-consistent True \( (\text{acquired True ?take-sb}_j \text{O}_j) \) \( (\text{flush ?take-sb}_j \text{m}_{sb}) \) ?drop-sb\(_j\).

from non-volatile-owned-or-read-only-drop [OF nvo-j]
have \( \text{nvo-j-drop}: \) non-volatile-owned-or-read-only True \( (\text{share ?take-sb}_j \text{S}_{sb}) \) \( (\text{acquired True ?take-sb}_j \text{O}_j) \) ?drop-sb\(_j\) by simp

from outstanding-refs-non-volatile-Read\(_sb\) all-acquired [OF rc this a-in-nvr]

have \( \text{a-owns-acq-ror}: \)
\( a \in \text{O}_j \cup \text{all-acquired sbj} \cup \text{read-only-reads (acquired True ?take-sb}_j \text{O}_j) \) ?drop-sb\(_j\)
by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in simp add: acquired-takeWhile-non-volatile-Write\(_sb\) )

have \( \text{a-unowned-j}: \) \( a \notin \text{O}_j \cup \text{all-acquired sbj} \)
  proof (cases \( a \in \text{O}_j \))
  case False with \( \text{a-not-acquired-j} \) show \( \text{?thesis} \) by auto
  next
  case True
  from all-shared-acquired-in \( \text{[OF True a-unshared]} \) \( \text{a-notin-owns-j} \)
  have False by auto thus \( \text{?thesis} .. \) qed
with \( \text{a-owns-acq-ror} \)
have \( \text{a-ror}: \) \( a \in \text{read-only (share ?take-sb}_j \text{S}_{sb}) \)
by auto

with \( \text{read-only-reads-unowned [OF j-bound'' i-bound neq-i-j [symmetric] ts}_{sb-j} ts_{sb-i}] \)
have \( \text{a-unowned-sb}: \) \( a \notin \text{O}_{sb} \cup \text{all-acquired sb} \)
by auto

from sharing-consis \( \text{[OF j-bound'' ts}_{sb-j} \text{]} \) sharing-consistent-append \( \text{[of S}_{sb} \text{O}_j ?take-sb}_j \text{ ?drop-sb}_j \) \( \text{?drop-sb}_j \)

have \( \text{consis-j-drop}: \) sharing-consistent \( (\text{share ?take-sb}_j \text{S}_{sb}) \) \( (\text{acquired True ?take-sb}_j \text{O}_j) \) ?drop-sb\(_j\) by auto

from \( \text{read-only-reads-read-only [OF nvo-j-drop consis-j-drop]} \) \( \text{a-ror a-unowned-j all-acquired-append [of ?take-sb}_j \text{ ?drop-sb}_j \text{]} \) \( \text{acquired-takeWhile-non-volatile-Write}_{sb} \) \( \text{[of sb}_j \text{O}_j] \)

have \( a \in \text{read-only (share ?take-sb}_j \text{S}_{sb}) \)
by (auto simp add: )
from \( \text{read-only-share-all-shared [OF this]} \) \( \text{a-unshared} \)
have \( a \in \text{read-only S}_{sb} \)
by fastforce

500
from read-only-unacquired-share [OF read-only-unowned [OF i-bound ts\_sb\_i]]
  weak-sharing-consis [OF i-bound ts\_sb\_i] this] a-unowned-sb
have a ∈ read-only (share sb S_{sb})
  by auto

with a-not-ro show False
  by simp
qed

with reads-consistent-mem-eq-on-non-volatile-reads [OF - subset-refl reads-consis-flush-m]
have reads-consistent True (acquired True ?take-sb_j O_j) ?m-A suspends_j
  by (auto simp add: suspends_j)

hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb_j O_j) ?m-A ys
  by (simp add: split-suspends_j reads-consistent-append)

from valid-history [OF j-bound” ts\_sb\_j]
have h-consis:
  history-consistent ?sb_j (hd-prog p_j (?take-sb_j @?suspends_j)) (?take-sb_j @suspends_j)
  apply (simp only: split-suspends_j [symmetric] suspends_j)
  apply simp
  done

have last-prog-hd-prog: last-prog (hd-prog p_j ?take-sb_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
proof
  from last-prog have last-prog p_j (?take-sb_j @?drop-sb_j) = p_j
    by simp
  from last-prog-hd-prog-append’ [OF h-consis] this
  have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
    by (simp only: split-suspends_j [symmetric] suspends_j)
  moreover
  have last-prog (hd-prog p_j (?take-sb_j @ suspends_j)) ?take-sb_j =
    last-prog (hd-prog p_j suspends_j) ?take-sb_j
    apply (simp only: split-suspends_j [symmetric] suspends_j)
    by (rule last-prog-hd-prog-append)
  ultimately show ?thesis
    by (simp add: split-suspends_j [symmetric] suspends_j)
qed

from valid-write-sops [OF j-bound” ts\_sb\_j]
have ∀ sop∈write-sops (?take-sb_j @?suspends). valid-sop sop
  by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain valid-sops-take: ∀ sop∈write-sops ?take-sb_j. valid-sop sop and
valid-sops-drop: ∀ sop∈write-sops ys. valid-sop sop
  apply (simp only: write-sops-append )
  apply auto
  done

501
from read-tmps-distinct [OF j-bound’’ ts\_ab\_j]

have distinct-read-tmps (?)take-sb\_j@suspends\_j) by (simp add: split-suspends\_j [symmetric] suspends\_j)

then obtain
read-tmps-take-drop: read-tmps ?take-sb\_j intersection read-tmps suspends\_j = {}

apply (simp only: split-suspends\_j [symmetric] suspends\_j)
apply (simp only: distinct-read-tmps-append)
done

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]

have hist-consis': history-consistent \( \theta \_sb \_j \) (hd-prog p\_j suspends\_j) suspends\_j
by (simp add: split-suspends\_j [symmetric] suspends\_j)

from reads-consistent-drop-volatile-writes-no-volatile-reads [OF reads-consis]

have no-vol-read: outstanding-refs is-volatile-Read\_sb\_ys = {}
by (auto simp add: outstanding-refs-append suspends\_j [symmetric]
split-suspends\_j)

from flush-store-buffer-append [OF j-bound’’ is\_j [simplified split-suspends\_j] cph [simplified suspends\_j]

hist-consis' [simplified split-suspends\_j] valid-sops-drop distinct-read-tmps-drop [simplified split-suspends\_j]

no-volatile-Read\_ab-volatile-reads-consistent [OF no-vol-read], where
S = \(?share-A\)

obtain is\_j' R\_j' where
is\_j': instrs (Write\_sb True a' sop v A' L' R' W'# zs) @ is\_sb\_j =
is\_j' @ prog-instrs (Write\_sb True a' sop v A' L' R' W'# zs) and

steps-ys: (?ts-A\_j, ?m-A, ?share-A) \( \Rightarrow_d^* \)

(?ts-A\_j):= (last-prog (hd-prog p\_j (Write\_sb True a' sop v A' L' R' W'# zs))) ys,
is\_j',
\( \theta \_sb\_j \mid (dom \theta \_sb\_j − read-tmps (Write\_sb True a' sop v A' L' R' W' # zs)).(),)

D\_j \lor outstanding-refs is-volatile-Write\_sb ys \( \neq \{\}, \) acquired True ys
(acquired True ?take-sb\_j O\_j),R\_j' ],

flush ys ?m-A,
share ys ?share-A)

(is (_,_,_) \( \Rightarrow_d^* \) (?ts-ys,?m-ys,?shared-ys))
by (auto)

note conflict-computation = rtranclp-trans [OF rtranclp-r-rtranclp [OF steps-flush-sb, OF store-step] steps-ys]

from cph

have causal-program-history is\_sb\_j ((ys @ [Write\_sb True a' sop v A' L' R' W']) @ zs)
by simp

from causal-program-history-suffix [OF this]

have cph': causal-program-history is\_sb\_j zs.
interpret causal\_j: causal-program-history is\_sb\_j zs by (rule cph')

from causal\_j, causal-program-history [of [], simplified, OF refl] is\_j'
obtain is'"
  where is'\_j: is'\_j' = Write True a' sop A' L' R' W'\#is'\_j'' and
  is'\_j'' instrs zs @ is\_sb\_j = is'\_j'' @ prog-instrs zs
by clarsimp

from j-bound''
have j-bound-ys: j < length ?ts-ys
  by auto

from j-bound-ys neq-i-j
have ts-ys-j: ?ts-ys\_j!=(last-prog (hd-prog p\_j (Write\_sb True a' sop v A' L' R' W'\# zs)))
ys, is'\_j',
  \_sb\_j \_i (dom \_sb\_j - read-tmps (Write\_sb True a' sop v A' L' R' W'\# zs)),(),
  D_\_j \lor outstanding-refs is-volatile-Write\_sb ys \neq \{\},
  acquired True ys (acquired True ?take-sb\_j O_\_j),R_\_j')
by auto

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified is'\_j]
have a-unowned: (?ts-ys,?m-ys,?shared-ys).
proof
  { fix \_j O_\_j \_sb\_j
    assume j-bound: j < length (map owned ts\_sb)
    assume neq-i-j: i\neq j
    assume ts\_sb-j: (map O-sb ts\_sb)!j = (O_\_j,\_sb\_j)
    assume conflict: A \land all-acquired sb\_j \neq \{}
    have False
      by (auto simp add: Let-def)
  }
  qed

thus ?thesis
by (auto simp add: Let-def)

have A-unquired-by-others:
\forall j<length (map O-sb ts\_sb). i \neq j
  (let (O_\_j, sb\_j) = map O-sb ts\_sb! j
   in A \land all-acquired sb\_j = \{\})
proof -
{
  fix \_j O_\_j \_sb\_j
  assume j-bound: j < length (map owned ts\_sb)
  assume neq-i-j: i\neq j
  assume ts\_sb-j: (map O-sb ts\_sb)!j = (O_\_j,\_sb\_j)
  assume conflict: A \land all-acquired sb\_j \neq \{}
  have False
}

503
proof
  from j-bound leq
  have j-bound': j < length (map owned ts)
    by auto
  from j-bound have j-bound'': j < length ts_{sb}
    by auto
  from j-bound' have j-bound'''': j < length ts
    by simp

  from conflict obtain a' where
    a'-in: a' ∈ A and
    a'-in-j: a' ∈ all-acquired sbj
    by auto

  let ?take-sb_{j} = (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_{j})
  let ?drop-sb_{j} = (dropWhile (Not ◦ is-volatile-Write_{sb}) sb_{j})

  from ts-sim [rule-format, OF j-bound'''] ts_{sb}-j j-bound''
  obtain p_{j} suspends_{j} is_{sbj} D_{sbj} D_{j} R_{j} θ_{sbj} is_{j} where
    ts_{sb-j}: ts_{sb} ! j = (p_{j}, is_{sbj}, θ_{sbj}, sb_{j}, D_{sbj}, O_{j}, R_{j}) and
    suspends_{j}: suspends_{j} = (?drop-sb_{j} and
    is_{j}: instrs suspends_{j} @ is_{sbj} = is_{j} @ prog-instrs suspends_{j} and
    D_{j}: D_{sbj} = (D_{j} ∨ outstanding-refs is-volatile-Write_{sb} sb_{j} ≠ {}) and
    ts_{j}: ts_{j} = (hd-prog p_{j} suspends_{j}, is_{j},
      θ_{sbj} |' (dom θ_{sbj} − read-tmps suspends_{j}),(),
      D_{j}, acquired True ?take-sb_{j} O_{j}, release ?take-sb_{j} (dom S_{sb}) R_{j})
  apply (cases ts_{sb}'{j})
  apply (force simp add: Let-def)
  done

  from a'-in-j all-acquired-append [of ?take-sb_{j} ?drop-sb_{j}]
  have a' ∈ all-acquired ?take-sb_{j} ∨ a' ∈ all-acquired suspends_{j}
    by (auto simp add: suspends_{j})
  thus False
  proof
    assume a' ∈ all-acquired ?take-sb_{j}
    with A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb}'{j} j-bound a'-in
    show False
    by (auto dest: all-acquired-unshared-acquired)
  next
    assume conflict-drop: a' ∈ all-acquired suspends_{j}
    from split-all-acquired-in [OF conflict-drop]
    show False
    proof
      assume ∃ sop a'' v ys zs A L R W.
      suspends_{j} = ys @ Write_{sb} True a'' sop v A L R W# zs ∧ a' ∈ A
      then
      obtain a'' sop' v' ys zs A' L' R' W' where
        split-suspends_{j}: suspends_{j} = ys @ Write_{sb} True a'' sop' v' A' L' R' W'# zs
(is suspends_\_i = ?suspends) and
\( a' \_A', a' \in A' \)
by auto

from direct-memop-step.WriteVolatile [where \( \vartheta = \@ sb \) and \( m = \) flush ?drop-sb m]

have (Write True a (D, f) A L R W # is_{sb}'',
    \( \vartheta_{sb}, () \), flush ?drop-sb m ,\( D_{sb} \), acquired True sb \( O_{sb} \),
    release sb (dom \( S_{sb} \)) \( R_{sb} \),
    share ?drop-sb \( S \)
) \( \to \)
    (is_{sb}'', \( \vartheta_{sb}, () \), (flush ?drop-sb m)(a := f \( \vartheta_{sb} \)), True, acquired True sb \( O_{sb} \) \( A \) \( \to \) R,Map.empty,
    share ?drop-sb \( S \oplus R \ominus A \) L).

from direct-computation.concurrent-step.Memop [OF
i-bound-ts' [simplified is_{sb}] ts'_i [simplified is_{sb}] this [simplified is_{sb}]]

have store-step: (?ts', flush ?drop-sb m, share ?drop-sb \( S \)) \( \Rightarrow_d \)
    (?ts'[i] := (P_{sb}, is_{sb}'',
    \( \vartheta_{sb}, () \),True, acquired True sb \( O_{sb} \) \( A \) \( \to \) R,Map.empty)),
    (flush ?drop-sb m)(a := f \( \vartheta_{sb} \)),share ?drop-sb \( S \oplus W \) R \( \ominus A \) L)
    (is - \( \Rightarrow_d \) (?ts-A, ?m-A, ?share-A))
by (simp add: is_{sb})

from i-bound' have i-bound'': i < length ?ts-A
by simp

from valid-program-history [OF j-bound'' ts_{sb}-j]

have causal-program-history is_{sbj} sbj,
then have cph: causal-program-history is_{sbj} ?suspends
apply –
apply (rule causal-program-history-suffix [where sb=?take-sb_{j}] )
apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
apply (simp add: split-suspends_{j})
done

from ts_{j} neq-i-j j-bound

have ts-A-j: ?ts-A] = (hd-prog p_{j} (ys @ Write_{sb} True a'" sop'v' A'L'R'W'# zs),
    \( i_{sj} \),
    \( \vartheta_{sbj} |' \) (dom \( \vartheta_{sbj} \) – read-tmps (ys @ Write_{sb} True a'" sop'v' A'L'R'W'# zs)), (), \( D_{j} \),
    acquired True ?take-sb_{j} \( O_{j} \),release ?take-sb_{j} (dom \( S_{sb} \)) \( R_{j} \)
by (simp add: split-suspends_{j})

from j-bound''' i-bound' neq-i-j have j-bound'''': j < length ?ts-A
by simp

from valid-last-prog [OF j-bound'' ts_{sb}-j] have last-prog: last-prog p_{j} sb_{j} = p_{j}.
then
have lp: last-prog p_{j} ?suspends = p_{j}

505
apply –
apply (rule last-prog-same-append [where sb=?take-sbj])
apply (simp only: split-suspendsj [symmetric] suspendsj)
apply simp
done

from valid-reads [OF j-bound" tsab-j]
have reads-consis: reads-consistent False Oj mab sbj.

from reads-consistent-flush-all-until-volatile-write [OF (valid-ownership-and-sharing Ssb tsab)
j-bound"
tsab-j reads-consis]
  have reads-consis-m: reads-consistent True (acquired True ?take-sbj Oj) m suspendsj
by (simp add: m suspendsj)

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound" neq-i-j tsab-i tsab-j]
  have outstanding-refs is-Writeab ?drop-sb \cap outstanding-refs is-non-volatile-Readab
suspendsj = {}
  by (simp add: suspendsj)
  from reads-consistent-flush-independent [OF this reads-consis-m]
  have reads-consis-flush-m: reads-consistent True (acquired True ?take-sbj Oj)
      (flush ?drop-sb m) suspendsj.

from a-unowned-others [rule-format, OF - neq-i-j j-bound tsab-j]
  obtain a-notin-owns-j: a \notin acquired True ?take-sbj Oj and a-unshared: a \notin all-shared ?take-sbj
  by auto
  from a-not-acquired-others [rule-format, OF - neq-i-j j-bound tsab-j]
  have a-not-acquired-j: a \notin all-acquired sbj
  by auto

from outstanding-non-volatile-refs-owned-or-read-only [OF j-bound" tsab-j]
  have nvo-j: non-volatile-owned-or-read-only False Ssb Oj sbj.

  have a-no-non-vol-read: a \notin outstanding-refs is-non-volatile-Readab ?drop-sbj
proof
assumption a-in-nvr:a \in outstanding-refs is-non-volatile-Readab ?drop-sbj

from reads-consistent-drop [OF reads-consis]
  have rc: reads-consistent True (acquired True ?take-sbj Oj) (flush ?take-sbj mab)
  ?drop-sbj.

from non-volatile-owned-or-read-only-drop [OF nvo-j]
  have nvo-j-drop: non-volatile-owned-or-read-only True (share ?take-sbj Sab)
    (acquired True ?take-sbj Oj)
  ?drop-sbj
  by simp

506
from outstanding-refs-non-volatile-Reads-all-acquired [OF rc this a-in-nvr]

have a-owns-acq-ror:
  a ∈ Oj ∪ all-acquired sbj ∪ read-only-reads (acquired True ?take-sbj Oj) ?drop-sbj
by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in
  simp add: acquired-takeWhile-non-volatile-Write)

have a-unowned-j: a /∈ Oj ∪ all-acquired sbj
proof (cases a ∈ Oj)
  case False with a-not-acquired-j show ?thesis by auto
next
  case True
  from all-shared-acquired-in [OF True a-unshared] a-notin-owns-j
  have False by auto thus ?thesis ..
qed

with a-owns-acq-ror
have a-ror: a ∈ read-only-reads (acquired True ?take-sbj Oj) ?drop-sbj
  by auto

with read-only-reads-unowned [OF j-bound′′ i-bound neq-i-j [symmetric] ts sb-i ts sb-j]
have a-unowned-sb: a /∈ O sb ∪ all-acquired sb
  by auto

from sharing-consis [OF j-bound′′ ts sb-j] sharing-consistent-append [of s sb Oj ?take-sbj
  ?drop-sb]
have consis-j-drop: sharing-consistent (share ?take-sbj s sb) (acquired True ?take-sbj Oj)
  ?drop-sb by auto

from read-only-reads-read-only [OF nvo-j-drop consis-j-drop] a-ror a-unowned-j
  all-acquired-append [of ?take-sbj ?drop-sb] acquired-takeWhile-non-volatile-Write
  [of sbj Oj]
have a ∈ read-only (share ?take-sbj s sb)
  by (auto)
from read-only-share-all-shared [OF this] a-unshared
have a ∈ read-only s sb
  by fastforce

from read-only-unacquired-share [OF read-only-unowned [OF i-bound ts sb-i]
  weak-sharing-consis [OF i-bound ts sb-i this] a-unowned-sb
have a ∈ read-only (share sb s sb)
  by auto

with a-not-ro show False
  by simp
qed

with reads-consistent-mem-eq-on-non-volatile-reads [OF - subset-refl
  reads-consis-flush-m]
have reads-consistent True (acquired True ?take-sbj Oj) ?m-A suspends
by (auto simp add: suspends$j$)

hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb$j$ O$j$) ?m-As
by (simp add: split-suspends$j$ reads-consistent-append)

from valid-history [OF j-bound″ ts$_{sb-j}$]
have h-consis:
history-consistent $\theta_{sbj}$ (hd-prog p$j$ (?take-sb$j$@suspends$j$)) (?take-sb$j$@suspends$j$)
apply (simp only: split-suspends$j$ [symmetric] suspends$j$)
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p$j$ sb$j$) ?take-sb$j$ = (hd-prog p$j$ suspends$j$)
proof −
from last-prog have last-prog p$j$ (?take-sb$j$@?drop-sb$j$) = p$j$
by simp
from last-prog-hd-prog-append′ [OF h-consis] this
have last-prog (hd-prog p$j$ suspends$j$) ?take-sb$j$ = hd-prog p$j$ suspends$j$
by (simp only: split-suspends$j$ [symmetric] suspends$j$)
moreover
have last-prog (hd-prog p$j$ (?take-sb$j$ @ suspends$j$)) ?take-sb$j$ =
last-prog (hd-prog p$j$ suspends$j$) ?take-sb$j$
apply (simp only: split-suspends$j$ [symmetric] suspends$j$)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends$j$ [symmetric] suspends$j$)
qed

from valid-write-sops [OF j-bound″ ts$_{sb-j}$]
have $\forall$ sop\in write-sops (?take-sb$j$@?suspends). valid-sop sop
by (simp add: split-suspends$j$ [symmetric] suspends$j$)
then obtain valid-sops-take: $\forall$ sop\in write-sops ?take-sb$j$. valid-sop sop and
valid-sops-drop: $\forall$ sop\in write-sops ys. valid-sop sop
apply (simp only: write-sops-append )
apply auto
done

from read-tmps-distinct [OF j-bound″ ts$_{sb-j}$]
have distinct-read-tmps (?take-sb$j$@suspends$j$)
by (simp add: split-suspends$j$ [symmetric] suspends$j$)
then obtain
read-tmps-take-drop: read-tmps ?take-sb$j$ \cap read-tmps suspends$j$ = {} and
distinct-read-tmps-drop: distinct-read-tmps suspends$j$
apply (simp only: split-suspends$j$ [symmetric] suspends$j$)
apply (simp only: distinct-read-tmps-append)
done
from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]

last-prog-hd-prog
  have hist-consis': history-consistent θ
  by (simp add: split-suspends j [symmetric] suspends)
  from reads-consistent-drop-volatile-writes-no-volatile-reads
  [OF reads-consis]
  have no-vol-read: outstanding-refs is-volatile-Read sbys
  = {}
  by (auto simp add: outstanding-refs-append suspends j
       [symmetric] split-suspends j)

from flush-store-buffer-append [OF j-bound''']
  have causal-program-history is sbj
  where is :=Write True a'' sop' v' A' L' R' W' # is
  by (clarsimp)
  from causal-program-history-suffix [OF this]
  have causal j causal-program-history is sbj zs
  by (rule cph')
  from causal j causal-program-history [of [], simplified, OF refl] is j'
  obtain is j''
  where is j': is'_j = Write True a'' sop' A' L' R' W'#is''
     and is''': intrans zs @ is sbj = is'' @ prog-intrans zs
  by clarsimp

from j-bound'''
  have j-bound-ys: j < length ?ts-ys

509
by auto

from j-bound-ys neq-i-j
have ts-ys-j: ?ts-ys!j = (last-prog (hd-prog p j (Write sb True a′′ sop′ v′ A′ L′ R′ W′# zs)) ys, isj′,
  \vartheta sbj | (dom \vartheta sbj - read-temps (Write sb True a′′ sop′ v′ A′ L′ R′ W′# zs)), D j
\lor outstanding-refs is-volatile-Write sb ys \neq \{\},
acquired True ys (acquired True ?take-sb j Oj, Rj′)
by auto

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys, ?m-ys, ?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified isj′]
have A′-unowned:
\forall i < length ?ts-ys. j \neq i \longrightarrow (let (O i) = map owned ?ts-ys!i in A′ \cap O i = \{\})
apply cases
apply (fastforce simp add: Let-def is sb)[+]
done
from a′-in a′-A′ A′-unowned [rule-format, of i] neq-i-j i-bound′ A-R
show False
next
assume \exists A L R W ys zs.
suspends j = ys @ Ghost sb A L R W # zs \land a′ \in A
then
obtain ys zs A′ L′ R′ W′ where
split-suspends j: suspends j = ys @ Ghost sb A′ L′ R′ W′# zs (is suspends j = ?suspends)
and
a′-A′: a′ \in A′
by auto

from direct-memop-step.WriteVolatile [where \vartheta = \vartheta sb and m=flush ?drop-sb m]
have (Write True a (D, f) A L R W # is sb′,
  \vartheta sb, (), flush ?drop-sb m, D sb, acquired True sb O sb,
  release sb (dom S sb) R sb,
  share ?drop-sb S) \Rightarrow
  (is sb′, \vartheta sb, (), (flush ?drop-sb m)(a := f \vartheta sb), True, acquired True sb O sb \cup A - R, Map.empty,
  share ?drop-sb S \oplus W R \ominus A L).

from direct-computation.concurrent-step.Memop [OF i-bound-ts′ [simplified is sb] ts′-i [simplified is sb] this [simplified is sb]]
have store-step: (?ts′, flush ?drop-sb m, share ?drop-sb S) \Rightarrow d
  (?ts′ [i := (p sb, is sb′,
  \vartheta sb, (), True, acquired True sb O sb \cup A - R, Map.empty)],
  (flush ?drop-sb m)(a := f \vartheta sb, share ?drop-sb S \oplus W R \ominus A L)
(is - \Rightarrow d (?ts-A, ?m-A, ?share-A))
by (simp add: is sb)
from i-bound' have i-bound'": i < length ?ts-A
by simp

from valid-program-history [OF j-bound" tsA-j]
have causal-program-history sbj sbj.
then have cph: causal-program-history sbj ?suspends
apply —
apply (rule causal-program-history-suffix [where sb=?take-sb] )
apply (simp only: split-suspends [symmetric] suspends)
apply (simp add: split-suspends)
done

from ts_j neq-i-j j-bound
have ts-A-j: ?ts-A!j = (hd-prog p_j (ys @ Ghost sb A' L' R' W'# zs), isj, 
\hat{sbj} \cap (dom \hat{sbj} - read-tmps (ys @ Ghost sb A' L' R' W'# zs)), (), DJ, 
acquired True ?take-sbj O_j, release ?take-sbj (dom S(sb) R_j)
by (simp add: split-suspends)

from j-bound" i-bound' neq-i-j have j-bound"": j < length ?ts-A
by simp

from valid-last-prog [OF j-bound" tsA-j] have last-prog: last-prog p_j sb_j = p_j.
then
have lp: last-prog p_j ?suspends = p_j
apply —
apply (rule last-prog-same-append [where sb=?take-sb] )
apply (simp only: split-suspends [symmetric] suspends)
apply simp
done

from valid-reads [OF j-bound" tsA-j]
have reads-consis: reads-consistent False O_j mA sb_j.

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing 
S sb tsA]
  j-bound"
  tsA-j reads-consis]
have reads-consis-m: reads-consistent True (acquired True ?take-sbj O_j) m suspends
by (simp add: m suspends)

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound" neq-i-j 
  tsA-i tsA-j]
have outstanding-refs is-Write sb \cap outstanding-refs is-non-volatile-Read sb suspends = { }
by (simp add: suspends)
from reads-consistent-flush-independent [OF this reads-consis-m]
  have reads-consis-flush-m: reads-consistent True (acquired True ?take-sbj O_j)
  (flush ?drop-sb m) suspends_j.
from a-unowned-others [rule-format, OF - neq-i-j] j-bound ts_{sb-j}
obtain a-notin-owns-j: a \notin \text{acquired} \ ?\text{take-sb}_j \ O_j \ \text{and} \ a-unshared: \ a \notin \text{all-shared} \ ?\text{take-sb}_j

\text{by} \ \text{auto}

from a-not-acquired-others [rule-format, OF - neq-i-j] j-bound ts_{sb-j}
have a-not-acquired-j: \ a \notin \text{all-acquired} \ \text{sb}_j
\text{by} \ \text{auto}

from outstanding-non-volatile-refs-owned-or-read-only [OF j-bound'' ts_{sb-j}]
have nvo-j: \ \text{non-volatile-owned-or-read-only} \ \text{False} \ S_{sb} \ O_j \ \text{sb}_j.

have a-no-non-vol-read: \ a \notin \text{outstanding-refs is-non-volatile-Read}_{sb} \ ?\text{drop-sb}_j
proof
assume a-in-nvr:a \in \text{outstanding-refs is-non-volatile-Read}_{sb} \ ?\text{drop-sb}_j
from reads-consistent-drop [OF reads-consis]
have rc: \ \text{reads-consistent} \ \text{True} \ (\text{acquired True} \ ?\text{take-sb}_j \ O_j) \ (\text{flush } ?\text{take-sb}_j \ m_{sb})
?\text{drop-sb}_j.

from non-volatile-owned-or-read-only-drop [OF nvo-j]
have nvo-j-drop: \ \text{non-volatile-owned-or-read-only} \ \text{True} \ (\text{share } ?\text{take-sb}_j \ S_{sb})
 \text{(acquired True} ?\text{take-sb}_j \ O_j)
 ?\text{drop-sb}_j
\text{by} \ \text{simp}

from outstanding-refs-non-volatile-Read_{sb-all-acquired} [OF rc this a-in-nvr]

have a-owns-acq-ror:
a \in O_j \cup \text{all-acquired} \ \text{sb}_j \cup \text{read-only-reads} \ \text{(acquired True} \ ?\text{take-sb}_j \ O_j) \ ?\text{drop-sb}_j
\text{by} \ \text{(auto dest!}: \ \text{acquired-all-acquired-in} \ \text{all-acquired-takeWhile-dropWhile-in}
\text{simp add:} \ \text{acquired-takeWhile-non-volatile-Write}_{sb})

have a-unowned-j: \ a \notin O_j \cup \text{all-acquired} \ \text{sb}_j
proof \ \text{(cases} \ a \in O_j)
case \ False \ \text{with} \ a-not-acquired-j \ \text{show} \ ?\text{thesis} \ \text{by} \ \text{auto}
next
case \ True
\text{from} \ \text{all-shared-acquired-in} \ [OF \ \text{True a-unshared}] \ a-notin-owns-j
\text{have} \ False \ \text{by} \ \text{auto} \ \text{thus} \ ?\text{thesis} \ ..
qed

with a-owns-acq-ror
have a-ror: \ a \in \text{read-only-reads} \ \text{(acquired True} ?\text{take-sb}_j \ O_j) \ ?\text{drop-sb}_j
\text{by} \ \text{auto}

with \ \text{read-only-reads-unowned} \ [OF \ j-bound'' i-bound neq-i-j [symmetric] ts_{sb-j} ts_{sb-i}]
have a-unowned-sb: \ a \notin O_{sb} \cup \text{all-acquired} \ \text{sb}
\text{by} \ \text{auto}
from sharing-consis [OF j-bound'" ts_{sb-j}] sharing-consistent-append [of S_{sb} O_j ?take-sb_j ?drop-sb_j]
have consis-j-drop: sharing-consistent (share ?take-sb_j S_{sb}) (acquired True ?take-sb_j O_j) ?drop-sb_j
  by auto

from read-only-reads-read-only [OF nvo-j-drop consis-j-drop] a-ror a-unowned-j
  all-acquired-append [of ?take-sb_j ?drop-sb_j] acquired-takeWhile-non-volatile-Write_{sb}
  [of sb_j O_j]
have a ∈ read-only (share ?take-sb_j S_{sb})
  by (auto)
from read-only-share-all-shared [OF this] a-unshared
have a ∈ read-only S_{sb}
  by fastforce

with a-not-ro show False
  by simp
  qed

  with reads-consistent-mem-eq-on-non-volatile-reads [OF - subset-refl
  reads-consis-flush-m]
  have reads-consistent True (acquired True ?take-sb_j O_j) ?m-A suspends_j
  by (auto simp add: suspends_j)

  hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb_j O_j) ?m-A ys
  by (simp add: split-suspends_j reads-consistent-append)

from valid-history [OF j-bound" ts_{sb-j}]
have h-consis:
  history-consistent @_{sb_j} (hd-prog p_j (?take-sb_j@suspends_j)) (?take-sb_j@suspends_j)
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
proof
  from last-prog have last-prog p_j (?take-sb_j@?drop-sb_j) = p_j
    by simp
  from last-prog-hd-prog-append' [OF h-consis] this
  have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
    by (simp only: split-suspends_j [symmetric] suspends_j)
moreover
have last-prog (hd-prog p j (?take-sb j @ suspends j)) ?take-sb j =
last-prog (hd-prog p j suspends j) ?take-sb j
apply (simp only: split-suspends j [symmetric] suspends j)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends j [symmetric] suspends j)
qed

from valid-write-sops [OF j-bound" ts sb-j]
have ∀ sop∈write-sops (?take-sb j @ suspends j). valid-sop sop
by (simp add: split-suspends j [symmetric] suspends j)
then obtain valid-sops-take: ∀ sop∈write-sops ?take-sb j. valid-sop sop
and valid-sops-drop: ∀ sop∈write-sops ys. valid-sop sop
apply (simp only: write-sops-append )
apply auto
done

from read-tmps-distinct [OF j-bound" ts sb-j]
have distinct-read-tmps (?take-sb j @ suspends j)
by (simp add: split-suspends j [symmetric] suspends j)
then obtain read-tmps-take-drop: read-tmps ?take-sb j ∩ read-tmps suspends j = {} and
distinct-read-tmps-drop: distinct-read-tmps suspends j
apply (simp only: split-suspends j [symmetric] suspends j)
apply (simp only: distinct-read-tmps-append)
done

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]

last-prog-hd-prog
have hist-consis': history-consistent θ sbj (hd-prog p j suspends j) suspends j
by (simp add: split-suspends j [symmetric] suspends j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read sb ys = {}
by (auto simp add: outstanding-refs-append suspends j [symmetric]
split-suspends j )

from flush-store-buffer-append [OF j-bound" is j" is sb-j]
cph [simplified suspends j]
ts-A-j [simplified split-suspends j]
refl lp [simplified split-suspends j]
reads-consis-m-A-ys
hist-consis' [simplified split-suspends j]
valid-sops-drop distinct-read-tmps-drop
[split-suspends j]
no-volatile-Read sb-volatile-reads-consistent [OF no-vol-read], where
S = ?share-A]

obtain is j' R j' where
is j': instrs (Ghost sb A' L' R' W' # zs) @ is j' sbj =
is j'@ prog-instrs (Ghost sb A' L' R' W' # zs) and
steps-ys: (?ts-A, ?m-A, ?share-A) ⇒ d*
(?ts-A j:= (last-prog (hd-prog p j (Ghost sb A' L' R' W' # zs))) ys,
\( \text{is}_j', \quad \text{\( \theta \}\text{sbj} \mid (\text{dom} \\text{\( \theta \}\text{sbj} - \text{read-tmps (Ghost}_{sb} A' L' R' W'\# \text{zs})},()) \),} \\
\( \text{D}_j \vee \text{outstanding-refs is-volatile-Write}_{sb} \text{ys} \neq \{\}, \text{acquired True ys (acquired} \\
\text{True take-sb}_j \text{O}_j),\text{R}_j' \},} \\
\text{flush} \text{ys (\text{m-A},} \\
\text{\text{\( \text{\( \theta \}\text{sbj} \mid \text{\text{\( \theta \}\text{sbj} - read-tmps (Ghost}_{sb} \text{True a'' sop'}\nu' A' L' R' W'\# \text{zs})}.())}, \text{D}_j} \\
\vee \text{outstanding-refs is-volatile-Write}_{sb} \text{ys} \neq \{\}, \text{acquired True ys (acquired} \\
\text{True take-sb}_j \text{O}_j),\text{R}_j' \},} \\
\text{by (auto)} \\
\text{note conflict-computation = rtranclp-trans [OF rtranclp-r-rtranclp [OF} \\
\text{steps-flush-sb, OF store-step] steps-ys]} \\
\text{from cph} \\
\text{have causal-program-history is}_{sbj} ((\text{ys} @ [\text{Ghost}_{sb} A' L' R' W']) @ \text{zs})} \\
\text{by simp} \\
\text{from causal-program-history-suffix [OF this]} \\
\text{have cph': causal-program-history is}_{sbj} \text{zs.} \\
\text{interpret causal}_{sbj}: \text{causal-program-history is}_{sbj} \text{zs by (rule cph')} \\
\text{from causal}_{sbj}. \text{causal-program-history [of [], simplified, OF refl]} \text{is}_{j'} \\
\text{obtain is}_{j''} \\
\text{where is}_{j'}: \text{is}_{j'} = \text{Ghost A' L' R' W'\#is}_{j''} \text{and} \\
\text{is}_{j''}: \text{instrs zs @ is}_{sbj} = \text{is}_{j''} @ \text{prog-instrs zs} \\
\text{by clarsimp} \\
\text{from j-bound'''} \\
\text{have j-bound-ys: j < length ?ts-ys} \\
\text{by auto} \\
\text{from j-bound-ys neq-i-j} \\
\text{have ts-ys-j: \text{?ts-ys}_j'= (last-prog (hd-prog p}_j (\text{Ghost}_{sb} A' L' R' W'\# \text{zs})) \text{ys}, is}_{j'}', \\
\text{\( \text{\( \theta \}\text{sbj} \mid (\text{dom} \text{\( \theta \}\text{sbj} - \text{read-tmps (Write}_{sb} \text{True a'' sop'}\nu' A' L' R' W'\# \text{zs})}.())}, \text{D}_j \\
\vee \text{outstanding-refs is-volatile-Write}_{sb} \text{ys} \neq \{\}, \text{acquired True ys (acquired} \\
\text{True take-sb}_j \text{O}_j),\text{R}_j' \},} \\
\text{by auto} \\
\text{from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]} \\
\text{have safe-delayed (\text{?ts-ys,s-m-ys,s-shared-ys}).} \\
\text{from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified is}_{j'} \\
\text{have A'-unowned:} \\
\forall i < \text{length ?ts-ys. j}\neq i \longrightarrow (\text{let (O}_i) = \text{map owned ?ts-ys}_i \text{in A' \cap O}_i = \{\})} \\
\text{apply cases} \\
\text{apply (fastforce simp add: Let-def is}_{sbj}+)
\text{done} \\
\text{from a' in a'A' A'-unowned [rule-format, of i] neq-i-j i-bound' A-R} \\
\text{show False} \\
\text{by (auto simp add: Let-def)} \\
\text{qed} \\
\text{qed} \\
\text{qed}
thus ?thesis
by (auto simp add: Let-def)
qed

have A-no-read-only-reads-by-others:
\[ \forall j < \text{length} (\text{map } \mathcal{O}\text{-}sb \, \text{ts}_j) \, \, i \neq j \rightarrow \]
\[ (\text{let } (\mathcal{O}_j, \text{sb}_j) = \text{map } \mathcal{O}\text{-}sb \, \text{ts}_j! j \]
in \( A \cap \text{read-only-reads} (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \, \text{sb}_j) \mathcal{O}_j) \)
\[ (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \, \text{sb}_j) = \{\} \]

proof –

\{ fix j \mathcal{O}_j \text{ sb}_j \}
assume j-bound: j < \text{length} (\text{map } \mathcal{O}\text{-}sb \, \text{ts}_j)
assume neq-i-j: i \neq j
assume ts\_sb\_j: (\text{map } \mathcal{O}\text{-}sb \, \text{ts}_j)!j = (\mathcal{O}_j, \text{sb}_j)
let ?take-sb\_j = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \, \text{sb}_j)
let ?drop-sb\_j = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \, \text{sb}_j)

assume conflict: \( A \cap \text{read-only-reads} (\text{acquired True } ?\text{take-sb}_j \mathcal{O}_j) \) ?\text{drop-sb}_j \neq \{\}

have False

proof –
from j-bound leq
have j-bound': j < \text{length} (\text{map owned ts})
  by auto
from j-bound have j-bound'': j < \text{length} ts\_sb
  by auto
from j-bound' have j-bound''': j < \text{length} ts
  by simp

from conflict obtain a' where
  a'-in: a' \in A and
  a'-in-j: a' \in \text{read-only-reads} (\text{acquired True } ?\text{take-sb}_j \mathcal{O}_j) \) ?\text{drop-sb}_j
  by auto

from ts-sim [rule-format, OF j-bound''] ts\_sb\_j j-bound''

obtain p\_j suspends\_j is\_sb\_j D\_sb\_j D\_j R\_j \_sb\_j \_sj where
  ts\_sb\_j: \text{ts}_j! j = (p\_j, \text{is}_\text{sbj}, \_sb\_j, \text{sb}_j, D\_sb\_j, \mathcal{O}_j, R\_j) and
  suspends\_j: suspends\_j = \?drop-sb\_j and
  is\_j: \text{instrs suspends}_j @ \text{is}_\text{sbj} = \text{is}_j @ \text{prog-intrs suspends}_j and
  D\_j: D\_sb\_j = (D\_j \lor \text{outstanding-refs is-volatile-Write}_{\text{sb}} \, \text{sb}_j \neq \{\}) and
  ts\_j: ts\_sj = (\text{hd-prog } p\_j \text{ suspends}_j, \_sj, \_sbj \mid (\text{dom } \_sbj - \text{read-tmps suspends}_j), (), D\_j, \text{acquired True } ?\text{take-sb}_j \mathcal{O}_j, \text{release}
?\text{take-sb}_j (\text{dom } S\_sb) R\_j)
  apply (cases ts\_sb\_lj)
  apply (force simp add: Let-def)
done
from split-in-read-only-reads [OF a'_in-j [simplified suspends_j [symmetric]]]

obtain t v ys zs where
  split-suspends_j: suspends_j = ys @ Read_ab False a' t v# zs (is suspends_j = ?suspends)

and
  a'_umacq: a' \notin acquired True ys (acquired True ?take-sb_j O_j)
  by blast

from direct-memop-step.WriteVolatile [where \( \emptyset = \emptyset_{sb} \) and \( m = \text{flush} \?\text{drop-sb} \; m \)]

have (Write True a (D, f) A L R W# is suspends_j = ?suspends)
  \( \emptyset \), flush ?drop-sb m, D_ab, acquired True sb O_ab,
  release sb (dom S_{sb}) R_{sb}, share ?drop-sb S \rightarrow
  (is_{sb}', \emptyset_{sb}, \emptyset, (flush ?drop-sb m)(a := f \emptyset_{sb}), True, acquired True sb O_{sb} \cup
  A - R, \text{Map}.empty,
  share ?drop-sb S \oplus W R \ominus A L).

from direct-computation.concurrent-step.Memop [OF i-bound-ts' [simplified is_{sb}] ts'\_i [simplified is_{sb}] this [simplified is_{sb}]]

have store-step: (?ts' \[i := (p_{sb}, is_{sb}', \emptyset_{sb}, \emptyset, True, acquired True sb O_{sb} \cup A - R, \text{Map}.empty)]
  \( \emptyset \), flush ?drop-sb m, share ?drop-sb S)
  \Rightarrow d
  \( \emptyset_{sb}' \), \( \emptyset_{sb} \), (flush ?drop-sb m)(a := f \emptyset_{sb}), \text{share} \?\text{drop-sb} S \oplus W R \ominus A L)
  (is - \Rightarrow d (?ts-A, ?m-A, ?share-A))
  by (simp add: is_{sb})

from i-bound' have i-bound'': i < length ?ts-A
  by simp

from valid-program-history [OF j-bound'' ts_{sb}-j]

have causal-program-history is_{sbj} sb_j.

then have cph: causal-program-history is_{sbj} ?suspends
  apply –
  apply (rule causal-program-history-suffix [where sb=?take-sb_j ] )
  apply (simp only: split-suspends_j [symmetric] suspends_j)
  apply (simp add: split-suspends_j)
  done

from ts\_j neq-i-j j-bound

have ts-A\_j: ?ts-A\_j = (hd-prog p_j (ys @ Read_ab False a' t v# zs), is_j,
  \( \emptyset_{sbj} \), (dom \( \emptyset_{sbj} \) - read-tmps (ys @ Read_ab False a' t v# zs)), () D_j,
  acquired True ?take-sb_j O_j, release ?take-sb_j (dom S_{ab}) R_j)
  by (simp add: split-suspends_j)

from j-bound''' i-bound' neq-i-j have j-bound'''': j < length ?ts-A
  by simp

from valid-last-prog [OF j-bound'' ts_{sb}-j] have last-prog: last-prog p_j sb_j = p_j.

then
have \( \text{lp: last-prog } p_j \) ?suspends = \( p_j \)

apply --

apply (rule last-prog-same-append [\textbf{where} sb=?take-sb])

apply (simp only: split-suspends \( j \) [symmetric] suspends \( j \))

apply simp

done

from valid-reads [OF \( j\)-bound'] ts\( sb^j \)]

have reads-consis: reads-consistent False \( O_j \) \( m_{sb} \) \( sb_j \).

from reads-consistent-flush-all-until-volatile-write [OF \( j\)-bound"
\( ts_{sb}\) \( j\) reads-consis]

have reads-consis-m: reads-consistent True (acquired True ?take-sb\( j \) \( O_j \)) \( m \) suspends \( j \)

by (simp add: \( m \) suspends \( j \))

from outstanding-non-write-non-volatile-reads-drop-disj [OF \( i\)-bound"
\( j\)-bound"
\( \text{neq-i-} ts_{sb}\) \( i \) \( ts_{sb}\) \( j \)]

have outstanding-reps is-Write\( sb \) ?drop-sb \( \cap \) outstanding-reps is-non-volatile-Read\( sb \)

suspends \( j \) = \{ \}

by (simp add: suspends \( j \))

from reads-consistent-flush-independent [OF this reads-consis-m]

have reads-consis-flush-m: reads-consistent True (acquired True ?take-sb\( j \) \( O_j \))

\( (\text{flush} ?\text{drop-sb} \( m \)) \) suspends \( j \).

from a-unowned-others [rule-format, OF \( j\)-bound"
\( \text{neq-i-} ts_{sb}\) \( i \) \( \text{neq-i-j} ts_{sb}\) \( i \) \( ts_{sb}\) \( j \)]

obtain a-notin-owns-j: a \( \notin \) acquired True ?take-sb\( j \) \( O_j \) and a-unshared: a \( \notin \) all-shared ?take-sb\( j \)

by auto

from a-not-acquired-others [rule-format, OF \( j\)-bound neq-i-j] j-bound ts\( sb\) \( j \)

have a-not-acquired-j: a \( \notin \) all-acquired sb\( j \)

by auto

from outstanding-non-volatile-refs-owned-or-read-only [OF \( j\)-bound"
\( ts_{sb}\) \( j \)]

have nvo-j: non-volatile-owned-or-read-only False \( S_{sb} \) \( O_j \) sb\( j \).

have a-no-non-volatile-read: a \( \notin \) outstanding-reps is-non-volatile-Read\( sb \) ?drop-sb\( j \)

proof

\begin{itemize}
\item assume a-in-nvr:a \( \in \) outstanding-reps is-non-volatile-Read\( sb \) ?drop-sb\( j \)
\end{itemize}

from reads-consistent-drop [OF reads-consis]

\begin{itemize}
\item have \( \text{rc: reads-consistent True (acquired True } \) ?take-sb\( j \) \( O_j \) \( (\text{flush } \) ?take-sb\( j \) \( m_{sb} \)) \) ?drop-sb\( j \).
\end{itemize}

from non-volatile-owned-or-read-only-drop [OF nvo-j]

have nvo-j-drop: non-volatile-owned-or-read-only True (share ?take-sb\( j \) \( S_{sb} \))

\( (\text{acquired True } \) ?take-sb\( j \) \( O_j \) \)

\( \) ?drop-sb\( j \)

by simp
from outstanding-refs-non-volatile-Readₜsb-all-acquired [OF rc this a-in-nvr]

have a-owns-acq-ror:
  a ∈ Oₗ ∪ all-acquired sbₗ ∪ read-only-reads (acquired True ?take-sbₗ Oₗ) ?drop-sbₗ
  by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in
  simp add: acquired-takeWhile-non-volatile-Writeₜsb)

have a-unowned-j: a /∈ Oₗ ∪ all-acquired sbₗ
  proof (cases a ∈ Oₗ)
    case False with a-not-acquired-j show ?thesis by auto
    next
    case True
    from all-shared-acquired-in [OF True a-unshared] a-notin-owns-j
    have False by auto thus ?thesis ..
  qed

with a-owns-acq-ror
have a-ror: a ∈ read-only-reads (acquired True ?take-sbₗ Oₗ) ?drop-sbₗ
  by auto

with read-only-reads-unowned [OF j-bound" i-bound neq-i-j [symmetric] tsₜsb-j tsₜsb-i]
have a-unowned-sb: a /∈ Oₜsb ∪ all-acquired sb
  by auto

from sharing-consis [OF j-bound" tsₜsb-j] sharing-consistent-append [of Sₜsb Oₗ ?take-sbₗ
  ?drop-sbₗ]
  have consis-j-drop: sharing-consistent (share ?take-sbₗ Sₜsb) (acquired True ?take-sbₗ Oₗ) ?drop-sbₗ
  by auto

from read-only-reads-read-only [OF nvo-j-drop consis-j-drop] a-ror a-unowned-j
  [of sbₗ Oₗ]
  have a ∈ read-only (share ?take-sbₗ Sₜsb)
    by (auto)
  from read-only-share-all-shared [OF this] a-unshared
  have a ∈ read-only Sₜsb
    by fastforce

from read-only-unacquired-share [OF read-only-unowned [OF i-bound tsₜsb-i]
  weak-sharing-consis [OF i-bound tsₜsb-i] this] a-unowned-sb
  have a ∈ read-only (share sb Sₜsb)
    by auto

with a-not-ro show False
  by simp

qed

  with reads-consistent-mem-eq-on-non-volatile-reads [OF - subset-refl
  reads-consis-flush-m]

519
have reads-consistent True (acquired True ?take-sb \(O_j\)) \(?m\)-A suspends\(_j\)
by (auto simp add: suspends\(_j\))

hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb \(O_j\)) \(?m\)-A ys
by (simp add: split-suspends\(_j\) reads-consistent-append)

from valid-history [OF j-bound"] ts\(_{sb-j}\] have h-consis:
  history-consistent \(\vartheta_{sbj}\) (hd-prog \(p_j\) (?take-sb\(_j\)@suspends\(_j\))) (?take-sb\(_j\)@suspends\(_j\))
  apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
  apply simp
done

have last-prog-hd-prog: last-prog (hd-prog \(p_j\) sb\(_j\)) ?take-sb\(_j\) = (hd-prog \(p_j\) suspends\(_j\))
proof –
  from last-prog have last-prog \(p_j\) (?take-sb\(_j\)@?drop-sb\(_j\)) = \(p_j\)
  by simp
  from last-prog-hd-prog-append’ [OF h-consis] this
  have last-prog (hd-prog \(p_j\) suspends\(_j\)) ?take-sb\(_j\) = hd-prog \(p_j\) suspends\(_j\)
  by (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
  moreover
  have last-prog (hd-prog \(p_j\) (?take-sb\(_j\) @ suspends\(_j\))) ?take-sb\(_j\) =
  last-prog (hd-prog \(p_j\) suspends\(_j\)) ?take-sb\(_j\)
  apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
  by (rule last-prog-hd-prog-append)
  ultimately show \(?\)thesis
  by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
qed

from valid-write-sops [OF j-bound” ts\(_{sb-j}\)]
have \(\forall sop\in\)write-sops (?take-sb\(_j\)@?suspends\(_j\)). valid-sop sop
  by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
then obtain valid-sops-take: \(\forall sop\in\)write-sops ?take-sb\(_j\). valid-sop sop and
  valid-sops-drop: \(\forall sop\in\)write-sops ys. valid-sop sop
  apply (simp only: write-sops-append)
  apply auto
done

from read-tmps-distinct [OF j-bound” ts\(_{sb-j}\)]
have distinct-read-tmps (?take-sb\(_j\)@suspends\(_j\))
  by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
then obtain
  read-tmps-take-drop: read-tmps ?take-sb\(_j\) \(\cap\) read-tmps suspends\(_j\) = \{\} and
  distinct-read-tmps-drop: distinct-read-tmps suspends\(_j\)
  apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
  apply (simp only: distinct-read-tmps-append)
done

520
from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]
lst-prog-hd-prog
have hist-consis': history-consistent \( \partial_{sbj} (\text{hd-prog p}_j \text{suspends}_j) \) suspends_j
  by (simp add: split-suspends_j [symmetric] suspends_j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read sbys = {}
  by (auto simp add: outstanding-refs-append suspends_j [symmetric]
    split-suspends_j)

from flush-store-buffer-append [O
  OF j-bound'''' is_j [simplified split-suspends_j]
  cph [simplified suspends_j]
  ts-A-j [simplified split-suspends_j]
  refl lp [simplified split-suspends_j]
  reads-consis-m-A-ys
  hist-consis' [simplified split-suspends_j]
  valid-sops-drop distinct-read-tmps-drop
[O
  no-volatile-Read sb-volatile-reads-consistent [OF no-vol-read], where
S= share-A]
obtain is_j' R_j' where
  is_j': instrs (Read sb False a' t v# zs) @ is_sb j =
  is_j' @ prog-instrs (Read sb False a' t v# zs) and
  steps-ys: (?ts-A, ?m-A, ?share-A) \( \Rightarrow d^* \)
  (?ts-A_j:= (last-prog (hd-prog p_j (Read sb False a' t v# zs))) ys, is_j',
    \( \partial_{sbj} |' (\text{dom} \partial_{sbj} - \text{read-tmps} (Read sb False a' t v# zs)),(),
    \( D_j \lor \text{outstanding-refs is-volatile-Write sbys} \neq \{\}, \) acquired True ys
  (acquired True ?take-sb j O_j,R_j'),
  flush ys ?m-A,
  share ys ?share-A)
  (is (\_\_,\_) \( \Rightarrow d^* \) (?ts-ys,?m-ys,?shared-ys))
  by (auto)

note conflict-computation = rtranclp-trans [OF rtranclp-r-rtranclp [OF steps-flush-sb,
  OF store-step] steps-ys]
from cph
have causal-program-history is_sb j ((ys @ [Read sb False a' t v]) @ zs)
  by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history is_sb j zs.
interpret causal_j: causal-program-history is_sb j zs by (rule cph')

from causal_j.causal-program-history [of []], simplified, OF refl] is_j'
obtain is_j''
  where is_j'': is_j'' = Read False a' t#is_j'' and
  is_j'': instrs zs @ is_sb j = is_j'' @ prog-instrs zs
  by clarsimp

from j-bound'''
have j-bound-ys: j < length ?ts-ys
  by auto

521
from j-bound-ys neq-i-j
have ts-ys-j: ts-ys-j (last-prog (hd-prog p) (Read sb False a t v# zs)) ys, is-j',
    \( \check{\theta}_{sbj} \upharpoonright_{\cdot} (\text{dom } \check{\theta}_{sbj} - \text{read-tmps (Read sb False a t v# zs)}), () \),
\( D_j \lor \text{outstanding-refs is-volatile-}Write_{sb} \text{ ys } \neq \{ \} \),
acquired True ys (acquired True ?take-sb_j O_j), R_j'
by auto

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys, ?m-ys, ?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified is_j']
have a' \in acquired True ys (acquired True ?take-sb_j O_j) \lor
  a' \in read-only (share ys (share ?drop-sb S ⊕ W R ⊖ A L))
apply cases
apply (auto simp add: Let-def is_sb)
done

with a'-unacq
have a'-ro: a' \in read-only (share ys (share ?drop-sb S ⊕ W R ⊖ A L))
by auto
from a'-in
have a'-not-ro: a' \notin read-only (share ?drop-sb S ⊕ W R ⊖ A L)
by (auto simp add: in-read-only-convs)

have a' \in O_j \cup all-acquired sb_j

proof -
{
assume a-notin: a' \notin O_j \cup all-acquired sb_j
from weak-sharing-consis [OF j-bound'' ts-sb-j]
have weak-sharing-consistent O_j sb_j.
with weak-sharing-consistent-append [of O_j, ?take-sb_j, ?drop-sb_j]
have weak-sharing-consistent (acquired ?take-sb_j O_j) suspends_j
by (auto simp add: suspends_j)

with split-suspends_j
have weak-consis: weak-sharing-consistent (acquired ?take-sb_j O_j) ys
by (simp add: weak-sharing-consistent-append)
from all-acquired-append [of ?take-sb_j ?drop-sb_j]
have all-acquired ys \subseteq all-acquired sb_j
apply (clarsimp)
apply (clarsimp simp add: suspends_j [symmetric] split-suspends_j all-acquired-append)
done

with a-notin acquired-takeWhile-non-volatile-Write_{sb} [of sb_j O_j]
all-acquired-append [of ?take-sb_j ?drop-sb_j]
have a' \notin acquired True (takeWhile (Not \circ \text{is-volatile-}Write_{sb}) sb_j) O_j \cup all-acquired ys
by auto

from read-only-share-unowned [OF weak-consis this a'-'ro]
have a' \in read-only (share ?drop-sb S ⊕ W R ⊖ A L) .
with a'-not-ro have False
by auto
}
thus ?thesis by blast
qed

moreover
from A-unaquired-by-others [rule-format, OF j-bound neq-i-j] ts_{sb-j} j-bound
have A ∩ all-acquired sb_j = {}
  by (auto simp add: Let-def)
moreover
from A-unowned-by-others [rule-format, OF j-bound" neq-i-j] ts_{sb-j} j-bound
have A ∩ O_j = {}
  by (auto simp add: Let-def dest: all-shared-acquired-in)
moreover note a'\in
ultimately
show False
by auto
qed
}
thus ?thesis
by (auto simp add: Let-def)
qed

have valid-owner': valid-ownership S_{sb'} ts_{sb'}
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only S_{sb'} ts_{sb'}
proof –
from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb-i}]
have non-volatile-owned-or-read-only False S_{sb} O_{sb} (sb @ [Write_{sb} True a (D,f) (f @_{sb}) A L R W])
  by (auto simp add: non-volatile-owned-or-read-only-append)
from outstanding-non-volatile-refs-owned-or-read-only-only-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb'} sb' O_{sb'} S_{sb'})
qed

next
show outstanding-volatile-writes-unowned-by-others ts_{sb'}
proof (unfold-locales)
fix i_{1} j p_{1} i_{1} s_{1} O_{1} R_{1} D_{1} x_{1} s_{b_{1}} p_{j} i_{j} O_{j} R_{j} D_{j} x_{j} s_{b_{j}}
assume i_{1}-bound: i_{1} < length ts_{sb'}
assume j-bound: j < length ts_{sb'}
assume i_{1}=j: i_{1} \neq j
assume ts_{i-1}: ts_{sb'} i_{1} = (p_{1},i_{1},x_{1},s_{b_{1}},D_{1},O_{1},R_{1})
assume ts_{j}: ts_{sb'} i_{j} = (p_{j},i_{j},x_{j},s_{b_{j}},D_{j},O_{j},R_{j})
show (O_{j} ∪ all-acquired sb_{j}) ∩ outstanding-refs is-volatile-Write_{sb} sb_{1} = {}
proof (cases i_{1}=i)
case True
with i_{1}-j have i=j: i\neq j
  by simp

from j-bound have j-bound': j < length ts_{sb}
  by (simp add: ts_{sb}')

hence j-bound'': j < length (map owned ts_{sb})
  by simp

from ts-j i-j have ts-j': ts_{sb}!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
  by (simp add: ts_{sb}')

from a-unowned-others [rule-format, OF - i-j] i-j ts-j j-bound
obtain a-notin-j: a ∉ acquired True (takeWhile (Not ◦ is-volatile-Write sb) sb_j)
  and a-unshared: a ∉ all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb_j)
  by (auto simp add: Let-def ts_{sb}')

from a-not-acquired-others [rule-format, OF - i-j] i-j ts-j j-bound
have a-notin-acq: a ∉ all-acquired sb_j
  by (auto simp add: Let-def ts_{sb}')

from outstanding-volatile-writes-unowned-by-others
[OF i-bound j-bound' i-j ts_{sb}-i ts-j']
have (O_j ∪ all-acquired sb_j) ∩ outstanding-refs is-volatile-Write_{sb} sb = {}.

with ts-i_1 a-notin-j a-unshared a-notin-acq True i-bound show ?thesis
  by (auto simp add: ts_{sb}' sb' outstanding-refs-append acquired-takeWhile-non-volatile-Write_{sb} dest: all-shared-acquired-in)

next
  case False
  note i_1-i = this
  from i_1-bound have i_1-bound': i_1 < length ts_{sb}
    by (simp add: ts_{sb}')

from ts-i_1 False have ts-i_1': ts_{sb}!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)
  by (simp add: ts_{sb}')

show ?thesis
  proof (cases j=i)
    case True
    from i_1-bound'
    have i_1-bound'': i_1 < length (map owned ts_{sb})
      by simp

      from outstanding-volatile-writes-unowned-by-others
      [OF i_1-bound' i-bound i_1-i ts-i_1' ts_{sb}-i]
      have (O_{sb} ∪ all-acquired sb) ∩ outstanding-refs is-volatile-Write_{sb} sb_1 = {}.
      moreover
      from A-unused-by-others [rule-format, OF - False [symmetric]] False ts-i_1 i_1-bound
      have A ∩ outstanding-refs is-volatile-Write_{sb} sb_1 = {}
      by (auto simp add: Let-def ts_{sb}')

      ultimately
      show ?thesis
      using ts-j True ts_{sb}'
      by (auto simp add: i-bound ts_{sb}' O_{sb}' sb' all-acquired-append)

    next
    case False
    from j-bound have j-bound': j < length ts_{sb}
      by (simp add: ts_{sb}')

524
from ts-j False have ts-j': ts_{sb}'_j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
by (simp add: ts_{sb}')
from outstanding-volatile-writes-unowned-by-others
[OF i-bound' j-bound' i-bound' ts-i ts-j]
show (O_j ∪ all-acquired sb_j) ∩ outstanding-refs is-volatile-Write_{sb} sb_1 = \{\} .
qed
qed
qed
next
show ownership-distinct ts_{sb}'
proof −
  have ∀ j<length ts_{sb}. i ≠ j →
    (let (p_j, is_j, θ_j, sb_j, D_j, O_j, R_j) = ts_{sb} ! j
    in (O_{sb} ∪ all-acquired sb_j') ∩ (O_j ∪ all-acquired sb_j) = \{\})
  proof −
    { fix j p_j is_j O_j R_j D_j acq_j θ_j sb_j
      assume neq-i-j: i ≠ j
      assume j-bound: j < length ts_{sb}
      assume ts_{sb}'-j: ts_{sb} ! j = (p_j, is_j, θ_j, sb_j, D_j, O_j, R_j)
      have (O_{sb} ∪ all-acquired sb_j') ∩ (O_j ∪ all-acquired sb_j) = \{\}
      proof −
        { fix a'
          assume a'-in-i: a' ∈ (O_{sb} ∪ all-acquired sb_j')
          assume a'-in-j: a' ∈ (O_j ∪ all-acquired sb_j)
          have False
          proof −
            from a'-in-i have a' ∈ (O_{sb} ∪ all-acquired sb)
            by (simp add: sb' all-acquired-append)
            then show False
            proof
              assume a' ∈ (O_{sb} ∪ all-acquired sb)
              with ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}'-i ts_{sb}'-j] a'-in-j
              show \?thesis
            by auto
          next
          assume a' ∈ A
          moreover
          have j-bound': j < length (map owned ts_{sb})
          using j-bound by auto
          from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb}'-j j-bound
          obtain A ∩ acquired True (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j) O_j = \{\} and
          A ∩ all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j) = \{\}
          by (auto simp add: Let-def)
        } moreover
        from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb}'-j j-bound
        have A ∩ all-acquired sb_j = \{\}
        by auto
        ultimately
      }
show ?thesis
using a'-in-j
by (auto dest: all-shared-acquired-in)
qed
qed
}

then show ?thesis by auto
qed
}

then show ?thesis by (fastforce simp add: Let-def)
qed

from ownership-distinct-nth-update [OF i-bound ts' sb-i this]
show ?thesis by (simp add: ts' sb' O sb'' sb'')
qed

next
show read-only-reads-unowned ts' sb'
proof
fix n m
fix p_n is_n O_n R_n D_n \vartheta_n sb_n p_m is_m O_m R_m D_m \vartheta_m sb_m
assume n-bound: n < length ts' sb'
and m-bound: m < length ts' sb'
and neq-n-m: n\neq m
and nth: ts' sb'!n = (p_n, is_n, \vartheta_n, sb_n, D_n, O_n, R_n)
and mth: ts' sb'!m = (p_m, is_m, \vartheta_m, sb_m, D_m, O_m, R_m)
from n-bound have n-bound': n < length ts' sb' by (simp add: ts' sb')
from m-bound have m-bound': m < length ts' sb' by (simp add: ts' sb')

show (O_m \cup all-acquired sb_m) \cap
    read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write_{sb} sb_n) O_n)
    (dropWhile (Not \circ is-volatile-Write_{sb} sb_n)) sb_n) =
{}
proof (cases m=i)
case True
with neq-n-m have neq-n-i: n\neq i
by auto
with n-bound nth i-bound have nth': ts' sb'!n = (p_n, is_n, \vartheta_n, sb_n, D_n, O_n, R_n)
by (auto simp add: ts' sb')
note read-only-reads-unowned [OF n-bound' i-bound neq-n-i nth' ts' sb-i]
moreover
from A-no-read-only-reads-by-others [rule-format, OF - neq-n-i [symmetric]] n-bound'
nth'
have A \cap read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write_{sb} sb_n) O_n)
    (dropWhile (Not \circ is-volatile-Write_{sb} sb_n)) sb_n) =
{}
by auto
ultimately
show ?thesis
using True ts' sb-i nth' mth n-bound' m-bound'

526
by (auto simp add: ts_{sb}' \sigma_{sb}' \sigma_{sb}' all-acquired-append)

next

  case False
  note neq-m-i = this

  with m-bound mth i-bound have mth': ts_{sb}'!m = (p_m, is_m, \theta_m, s_m, D_m, \sigma_m, \sigma_m)
    by (auto simp add: ts_{sb}')

  show ?thesis
  proof (cases n=i)
    case True
    note read-only-reads-unowned [OF i-bound m-bound' neq-m-i [symmetric] ts_{sb}'!mth]
    then show ?thesis
      using True neq-m-i ts_{sb}'!mth
    done
    next
    case False
    with n-bound nth i-bound have nth': ts_{sb}'!n = (p_n, is_n, \theta_n, s_n, D_n, \sigma_n, \sigma_n)
      by (auto simp add: ts_{sb}')

    from read-only-reads-unowned [OF n-bound m-bound' neq-n-m nth' mth'] False neq-m-i
    show ?thesis
      by (clarsimp)

  qed

  qed

  have valid-hist': valid-history program-step ts_{sb}'
    proof
      from valid-history [OF i-bound ts_{sb}'!i]
      have history-consistent \theta_{sb} (hd-prog p_{sb} sb) sb.
      with valid-write-sops [OF i-bound ts_{sb}'!i] D-subset
      valid-implies-valid-prog-hd [OF i-bound ts_{sb}'!i valid]
      have history-consistent \theta_{sb} (hd-prog p_{sb} (sb@[\text{Write}_{sb} True a (D,f) (f \sigma_{sb}) A L R W]))
        (sb@[\text{Write}_{sb} True a (D,f) (f \sigma_{sb}) A L R W])
        apply 
        apply (rule history-consistent-appendII)
        apply (auto simp add: hd-prog-append-Write_{sb})
      done
      from valid-history-nth-update [OF i-bound this]
      show ?thesis by (simp add: ts_{sb}'!sb' \theta_{sb}')
      qed

  have valid-reads': valid-reads m_{sb} ts_{sb}'
    proof
      from valid-reads [OF i-bound ts_{sb}'!i]
      have reads-consistent False \sigma_{sb} m_{sb} sb .

  527
from reads-consistent-snoc-Write \( s_b \) \( [O F \ this] \)

have reads-consistent False \( \mathcal{O}_{s_b} m_{s_b} \) \( (s_b \oplus \{Write_{s_b} \ True \ a \ (D,f) (f \ \oplus_{s_b} A \ L \ R \ W)\}) \).

from valid-reads-nth-update \( [O F \ i-bound \ this] \)

show \( \theta \)thesis by (simp add: \textstyle \{ts_{s_b}, sb' \mathcal{O}_{s_b}\} \)

qed

have valid-sharing': valid-sharing \( S_{s_b}, ts_{s_b}' \)

proof (intro-locales)

from outstanding-non-volatile-writes-unshared \( [O F \ i-bound \ ts_{s_b}-i] \)

have non-volatile-writes-unshared \( S_{s_b} \) \( (s_b \oplus \{Write_{s_b} \ True \ a \ (D,f) (f \ \oplus_{s_b} A \ L \ R \ W)\}) \)

by (auto simp add: non-volatile-writes-unshared-append)

from outstanding-non-volatile-writes-unshared-nth-update \( [O F \ i-bound \ this] \)

show outstanding-non-volatile-writes-unshared \( S_{s_b}, ts_{s_b}' \)

by (simp add: \textstyle \{ts_{s_b}, sb' \mathcal{O}_{s_b}\} \)

next

from sharing-consis \( [O F \ i-bound \ ts_{s_b}-i] \)

have consis': sharing-consistent \( S_{s_b}, \mathcal{O}_{s_b}, sb \).

from A-shared-owned

have \( A \subseteq \dom (share \ ?\text{drop-sb} \mathcal{S}) \cup \text{acquired True } sb \mathcal{O}_{s_b} \)

by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write_{s_b})

moreover have \( dom (share \ ?\text{drop-sb} \mathcal{S}) \subseteq dom \mathcal{S} \cup \dom (\text{share sb } \mathcal{S}_{s_b}) \)

proof

fix \( a' \)

assume \( \text{a'-in: } a' \in \dom (share \ ?\text{drop-sb} \mathcal{S}) \)

from share-unshared-in \( [O F \ a'-in] \)

show \( a' \in \dom \mathcal{S} \cup \dom (\text{share sb } \mathcal{S}_{s_b}) \)

proof

assume \( a' \in \dom (\text{share sb } \mathcal{S}_{s_b}) \)

from share-mono-in \( [O F \ this] \) share-append \( [of ?\text{take-sb } ?\text{drop-sb}] \)

have \( a' \in \dom (\text{share sb } \mathcal{S}_{s_b}) \)

by auto

thus \( \theta \)thesis

by simp

next

assume \( a' \in \dom \mathcal{S} \land a' \notin \text{all-unshared } \text{drop-sb} \)

thus \( \theta \)thesis by auto

qed

qed

ultimately

have A-subset: \( A \subseteq \dom \mathcal{S} \cup \dom (\text{share sb } \mathcal{S}_{s_b}) \cup \text{acquired True } sb \mathcal{O}_{s_b} \)

by auto

with A-unowned-by-others

have \( A \subseteq \dom (\text{share sb } \mathcal{S}_{s_b}) \cup \text{acquired True } sb \mathcal{O}_{s_b} \)

proof

\{

fix \( x \)

assume \( x-A: x \in A \)

have \( x \in \dom (\text{share sb } \mathcal{S}_{s_b}) \cup \text{acquired True } sb \mathcal{O}_{s_b} \)

\}

528
proof
{
  assume x ∈ dom S

  from share-all-until-volatile-write-share-acquired OF (sharing-consis S sb ts sb)
  i-bound ts sb i this [simplified S]
  A-unowned-by-others x-A
  have ?thesis
  by (fastforce simp add: Let-def)
}

  with A-subset show ?thesis using x-A by auto
qed

thus ?thesis by blast
qed

with consis’ L-subset A-R R-acq
have sharing-consistent S sb O sb (sb @ [Write sb True a (D, f) (f \(≠ₜ sb\)) A L R W])
  by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write sb)
from sharing-consis-nth-update OF i-bound this
show sharing-consis S sb’ ts sb’
  by (simp add: ts sb’ O sb’ sb’ S sb’)
next
from read-only-unowned-nth-update OF i-bound read-only-unowned OF i-bound ts sb i]
show read-only-unowned S sb’ ts sb’
  by (simp add: S sb’ ts sb’ O sb’)
next
from unowned-shared-nth-update OF i-bound ts sb i subset-refl
show unowned-shared S sb’ ts sb’
  by (simp add: ts sb’ sb’ O sb’ S sb’)
next
from a-not-ro no-outstanding-write-to-read-only-memory OF i-bound ts sb i]
have no-write-to-read-only-memory S sb (sb @ [Write sb True a (D, f) (f \(≠ₜ sb\)) A L R W])
  by (simp add: no-write-to-read-only-memory-append)

from no-outstanding-write-to-read-only-memory-nth-update OF i-bound this
show no-outstanding-write-to-read-only-memory S sb’ ts sb’
  by (simp add: S sb’ ts sb’ sb’)
qed

  have tmps-distinct’ tmps-distinct ts sb’
proof (intro-locales)
from load-tmps-distinct OF i-bound ts sb i]
have distinct-load-tmps is sb’ by (simp add: is sb)
from load-tmps-distinct-nth-update OF i-bound this
show load-tmps-distinct ts sb’ by (simp add: ts sb’)
next
from read-tmps-distinct OF i-bound ts sb i]
have distinct-read-tmps (sb @ [Write sb True a (D, f) (f \(≠ₜ sb\)) A L R W])
  by (auto simp add: distinct-read-tmps-append)
from read-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct ts sb' by (simp add: ts sb')
next
from load-tmps-read-tmps-distinct [OF i-bound ts sb'-i]
have load-tmps is sb' ∩ read-tmps (sb @ [Write sb True a (D, f) (f, θ) A L R W]) = {}
  by (auto simp add: read-tmps-append is sb)
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct ts sb' by (simp add: is sb')
qed

have valid-sops': valid-sops ts sb'
proof
from valid-store-sops [OF i-bound ts sb'-i]
obtain valid-Df: valid-sop (D, f) and
  valid-store-sops': ∀ sop∈store-sops is sb'. valid-sop sop
  by (auto simp add: is sb)
from valid-Df valid-write-sops [OF i-bound ts sb'-i]
have valid-write-sops': ∀ sop∈write-sops (sb@ [Write sb True a (D, f) (f, θ) A L R W]). valid-sop sop
  by (auto simp add: write-sops-append)
from valid-sops-nth-update [OF i-bound valid-write-sops' valid-store-sops']
show ?thesis by (simp add: ts sb' sb')
qed

have valid-dd': valid-data-dependency ts sb'
proof
from data-dependency-consistent-instrs [OF i-bound ts sb'-i]
obtain D-indep: D ∩ load-tmps is sb' = {} and
  dd-is: data-dependency-consistent-instrs (dom θ sb') is sb'
  by (auto simp add: is sb θ sb')
from load-tmps-write-tmps-distinct [OF i-bound ts sb'-i] D-indep
have load-tmps is sb' ∩ (∪ (fst ' write-sops (sb@ [Write sb True a (D, f) (f, θ) A L R W]))) = {}
  by (auto simp add: write-sops-append is sb)
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show ?thesis by (simp add: ts sb' sb')
qed

have load-tmps-fresh': load-tmps-fresh ts sb'
proof
from load-tmps-fresh [OF i-bound ts sb'-i]
have load-tmps is sb' ∩ dom θ sb = {}
  by (auto simp add: is sb)
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis by (simp add: is sb')
qed

have enough-flushs': enough-flushs ts sb'
proof
from clean-no-outstanding-volatile-Write sb [OF i-bound ts sb'-i]
have \neg True \rightarrow \text{outstanding-refs is-volatile-Write}_{sb} (\text{sb}@[\text{Write}_{sb} \text{True a (D,f)} (f \varphi_{sb}) A L R W]) = \{} \\
by (auto simp add: outstanding-refs-append)
from \text{enough-flushs-nth-update} [\text{OF i-bound this}]
show \text{?thesis}
  by (simp add: ts_{sb}', sb', is_{sb}')
qed

have valid-program-history': valid-program-history ts_{sb}'
proof
from valid-program-history [OF i-bound ts_{sb}-i]
have causal-program-history is_{sb} sb .
then have causal': causal-program-history is_{sb}' (\text{sb}@[\text{Write}_{sb} \text{True a (D,f)} (f \varphi_{sb}) A L R W])
  by (auto simp: causal-program-history-Write is_{sb})
from valid-last-prog [OF i-bound ts_{sb}-i]
have last-prog p_{sb} sb = p_{sb}.
hence last-prog p_{sb} (\text{sb}@[\text{Write}_{sb} \text{True a (D,f)} (f \varphi_{sb}) A L R W]) = p_{sb}
  by (simp add: last-prog-append-Write_{sb})
from valid-program-history-nth-update [OF i-bound causal' this]
show \text{?thesis}
  by (simp add: ts_{sb}', sb')
qed

show \text{?thesis}
proof (cases outstanding-refs is-volatile-Write_{sb} sb = \{} )
case True
from True have flush-all: takeWhile (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
from True have suspend-nothing: dropWhile (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb = []
  by (auto simp add: outstanding-refs-conv)
hence suspends-empty: suspends = []
  by (simp add: suspends)
from suspends-empty is-sim have: is = Write True a (D,f) A L R W# is_{sb}'
  by (simp add: is_{sb})
with suspends-empty ts-i
have ts-i: ts!i = (p_{sb}, \text{Write True a (D,f)} A L R W# is_{sb}', \varphi_{sb}, (), D, \text{acquired True} ?\text{take-sb} \varnothing_{sb}, \text{release} ?\text{take-sb} (\text{dom} S_{sb}) R_{sb})
  by simp
have (ts, m, S) \Rightarrow_{d^*} (ts, m, S) by auto

moreover

note flush-commute =
  flush-all-until-volatile-write-append-volatile-write-commute

531
from True
have drop-app: dropWhile (Not ◦ is-volatile-Write sb) 
(sb@[Write sb True a (D,f) (f ◦sb) A L R W]) = 
[Write sb True a (D,f) (f ◦sb) A L R W]
by (auto simp add: outstanding-refs-conv)

have (ts sb ′,m sb ,S sb ′) ∼ (ts,m,S)
apply (rule sim-config.intros)
apply (simp add: m flush-commute ts sb ′ ◦ sb ′ O sb ′ ⊢ sb ′ ⊢ R sb ′ ⊢ sb ′)
using [OF i-bound ts sb ′ i [simplified is sb ]]
apply (clarsimp simp add: SS sb ′ ts sb ′ sb ′ O sb ′ R sb ′ θ sb ′)
using leq
apply (simp add: ts sb ′)
using i-bound i-bound′ ts-sim ts-i
apply (clarsimp simp add: Let-def nth-list-update drop-app

ultimately show ?thesis
using valid-own′ valid-hist′ valid-reads′ valid-sharing′ tmps-distinct′
valid-sops′
valid-dd′ load-tmp-fresh′ enough-flushs′
valid-program-history′ valid′ m sb ′ S sb ′
by auto
next

then obtain r where r-in: r ∈ set sb and volatile-r: is-volatile-Write sb r
by (auto simp add: outstanding-refs-conv)
from takeWhile-dropWhile-real-prefix
[OF r-in, of (Not ◦ is-volatile-Write sb), simplified, OF volatile-r]
obtain a′ v′ sb ′A″ L″ R″ W″ sop′ where
sb-split: sb = takeWhile (Not ◦ is-volatile-Write sb) sb @ Write sb True a′ sop′ v′ A″ L″
R″ W″# sb″
and
drop: dropWhile (Not ◦ is-volatile-Write sb) sb = Write sb True a′ sop′ v′ A″ L″ R″ W″ #
sb″
apply (auto)
subgoal for y ys
apply (case-tac y)
apply auto
done
done
from drop suspends have suspends: suspends = Write sb True a′ sop′ v′ A″ L″ R″ W″ #
sb″

532
by simp

have \((ts, m, S) \Rightarrow_d^* (ts, m, S)\) by auto

moreover

note flush-commute =
  flush-all-until-volatile-write-append-unflushed [OF False i-bound ts sb-i]

have Write sb True a' sop' v' A'' L'' R'' W'' \in set sb
  by (subst sb-split) auto
note drop-app = dropWhile-append1
[OF this, of (Not \circ is-volatile-Write sb), simplified]

have \((ts sb', m sb, S sb') \sim (ts, m, S)\)
  apply (rule sim-config).intros
  apply (simp add: m flush-commute ts sb' O sb' R sb' \theta sb' sb')
  using share-all-until-volatile-write-Write-commute
  [OF i-bound ts sb-i [simplified is sb]]
  apply (clarsimp simp add: SS sb' ts sb' sb' O sb' R sb' \theta sb' D sb' outstanding-refs-append takeWhile-tail release-append split: if-split-asm)
  done
ultimately show \(?thesis
  using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-dd'
  valid-sops' load-tmps-fresh' enough-flushs'
  valid-program-history' valid' m sb' S sb'
  by (auto simp del: fun-upd-apply )
qed
next
  case SBHFence
  then obtain
    is sb: is sb = Fence # is sb' and
    sb: sb=[] and
    O sb': O sb'=O sb and
    R sb': R sb'=Map.empty and
    \theta sb': \theta sb' = \theta sb and
    D sb': \neg D sb' and
    sb': sb'=sb and
    m sb': m sb' = m sb and
    S sb': S sb'=S sb
  by auto

  have valid-own': valid-ownership S sb' ts sb'

533
proof (intro-locales)

show outstanding-non-volatile-refs-owned-or-read-only \( S_{sb}' \) \( ts_{sb}' \)

proof –

have non-volatile-owned-or-read-only False \( S_{sb} \) \( O_{sb} '[[ \]
by simp
from outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this]
show ?thesis by (simp add: \( ts_{sb}' \) \( sb' \) \( sb \) \( O_{sb} ' S_{sb}' \))
qed

next
from outstanding-volatile-writes-unowned-by-others-store-buffer
[OF i-bound \( ts_{sb}' \) -i subset-refl]
show outstanding-volatile-writes-unowned-by-others \( ts_{sb}' \)
by (simp add: \( ts_{sb}' \) \( sb' \) \( sb \) \( O_{sb} ' S_{sb}' \))
next
from read-only-reads-unowned-nth-update [OF i-bound \( ts_{sb}' \) , of \( [] \) \( O_{sb} \)]
show read-only-reads-unowned \( ts_{sb}' \)
by (simp add: \( ts_{sb}' \) \( sb' \) \( sb \) \( O_{sb} ' S_{sb}' \))
next
from ownership-distinct-instructions-read-value-store-buffer-independent
[OF i-bound \( ts_{sb}' \) -i]
show ownership-distinct \( ts_{sb}' \)
by (simp add: \( ts_{sb}' \) \( sb' \) \( sb \) \( O_{sb} ' S_{sb}' \))
qed

have valid-hist': valid-history program-step \( ts_{sb}' \)
proof –
from valid-history [OF i-bound \( ts_{sb}' \) -i]
have history-consistent \( \theta_{sb} \) (hd-prog \( p_{sb} '[[ \]
by simp
from valid-history-nth-update [OF i-bound this]
show ?thesis by (simp add: \( ts_{sb}' \) \( sb' \) \( sb \) \( O_{sb} ' \theta_{sb}' \)))
qed

have valid-reads': valid-reads \( m_{sb} \) \( ts_{sb}' \)
proof –
have reads-consistent False \( O_{sb} m_{sb} '[] \)
by simp
from valid-reads-nth-update [OF i-bound this]
show ?thesis by (simp add: \( ts_{sb}' \) \( sb' \) \( sb \) \( O_{sb} ' \))
qed

have valid-sharing': valid-sharing \( S_{sb}' \) \( ts_{sb}' \)
proof (intro-locales)
have non-volatile-writes-unshared \( S_{sb} '[[ \]
by (simp add: sb)
from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared \( ts_{sb}' \)
by (simp add: \( ts_{sb}' \) \( sb' \) \( sb' S_{sb}' \))
next
have sharing-consistent \( S_{sb} O_{sb} '[[ \)
by simp

534
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis $S_{sb}'/ts_{sb}'$
  by (simp add: $ts_{sb}'/O_{sb}'/sb'/sb/S_{sb}'$
next
from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts_{sb}-i]]
show read-only-unowned $S_{sb}'/ts_{sb}'$
  by (simp add: $S_{sb}'/ts_{sb}'/O_{sb}'$
next
from unowned-shared-nth-update [OF i-bound ts_{sb}-i subset-refl]
show unowned-shared $S_{sb}'/ts_{sb}'$ by (simp add: $S_{sb}'/ts_{sb}'/O_{sb}'$/$S_{sb}'$
next
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound, of []]
show no-outstanding-write-to-read-only-memory $S_{sb}'/ts_{sb}'$
  by (simp add: $S_{sb}'/ts_{sb}'/sb'/sb$)
qed

have tmpos-distinct': tmpos-distinct ts_{sb}'
  proof (intro-locales)
from load-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-load-tmps is_{sb}'
  by (auto simp add: is_{sb} split: instr.splits)
from load-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-distinct ts_{sb}' by (simp add: $ts_{sb}'/sb'/sb/O_{sb}'/is_{sb}'$
next
from read-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-read-tmps [] by (simp add: $ts_{sb}'/sb'/sb/O_{sb}'$
from read-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct ts_{sb}' by (simp add: $ts_{sb}'/sb'/sb/O_{sb}'$
next
from load-tmps-read-tmps-distinct [OF i-bound ts_{sb}-i]
  load-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' $\cap$ read-tmps [] = {}
  by (clarsimp)
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct ts_{sb}' by (simp add: $ts_{sb}'/sb'/sb/O_{sb}'$
  qed

have valid-sops': valid-sops ts_{sb}'
  proof -
from valid-store-sops [OF i-bound ts_{sb}-i]
obtain
  valid-sops': $\forall sop\in$ store-sops is_{sb}'. valid-sop sop
  by (auto simp add: is_{sb} ts_{sb}'/sb'/sb/O_{sb}')
from valid-sops-nth-update [OF i-bound - valid-store-sops', where sb= []]
show ?thesis by (auto simp add: $ts_{sb}'/sb'/sb/O_{sb}'$
  qed

have valid-dd': valid-data-dependency ts_{sb}'

535
proof –
from data-dependency-consistent-instrs [OF i-bound ts\_sb\_i]
obtain
dd-is: data-dependency-consistent-instrs (\text{dom } \emptyset_{\text{sb}'} \cup \{\text{\text{fist\_sops ([]) = {}}\})
by (auto simp add: is_{\text{sb}} \emptyset_{\text{sb}}'
from load-tmps-write-tmps-distinct [OF i-bound ts\_sb\_i]
have load-tmps is_{\text{sb}'} \cap \{\text{\text{fist\_write-sops ([]) = {}}\})
by (auto simp add: write-sops-append)
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show \(\text{thesis}\) by (simp add: ts_{\text{sb}}' \emptyset_{\text{sb}}'
qed

have load-tmps-fresh': load-tmps-fresh ts_{\text{sb}}'
proof –
from load-tmps-fresh [OF i-bound ts\_sb\_i]
have load-tmps is_{\text{sb}'} \cap \{\text{\text{fist\_write-sops ([]) = {}}\})
by (auto simp add: is_{\text{sb}})
from load-tmps-fresh-nth-update [OF i-bound this]
show \(\text{thesis}\) by (simp add: is_{\text{sb}} ts_{\text{sb}}' \emptyset_{\text{sb}}'
qed

from enough-flushs-nth-update [OF i-bound, where sb=[]]
have enough-flushs': enough-flushs ts_{\text{sb}}'
by (auto simp add: ts_{\text{sb}}' sb' sb)

have valid-program-history': valid-program-history ts_{\text{sb}}'
proof –
have causal': causal-program-history is_{\text{sb}'} sb'
by (simp add: is_{\text{sb}} sb sb')
have last-prog p_{\text{sb}} sb' = p_{\text{sb}}
by (simp add: sb' sb)
from valid-program-history-nth-update [OF i-bound causal' this]
show \(\text{thesis}\) by (simp add: is_{\text{sb}} ts_{\text{sb}}' sb' sb' \emptyset_{\text{sb}}'
qed

from is-sim have is: is = Fence # is_{\text{sb}}'
by (simp add: suspends sb is_{\text{sb}})
with ts-i
have ts-i: tsli = (p_{\text{sb}}, Fence # is_{\text{sb}}', \emptyset_{\text{sb}}, (), D, acquired True ?take-sb O_{\text{sb}}, release ?take-sb (\text{dom } S_{\text{sb}}) R_{\text{sb}})
by (simp add: suspends sb)

from direct-memop-step.Fence
have (Fence # is_{\text{sb}}', \emptyset_{\text{sb}}, (), m, D, acquired True ?take-sb O_{\text{sb}}, release ?take-sb (\text{dom } S_{\text{sb}}) R_{\text{sb}}, S) \Rightarrow
(is_{\text{sb}}', \emptyset_{\text{sb}}, (), m, False, acquired True ?take-sb O_{\text{sb}}, Map.empty, S).
from direct-computation.concurrent-step.Memop [OF i-bound' ts-i this]
have (ts, m, S) \Rightarrow_d
moreover

have \((ts', m', S') \sim (ts := (p, is', \theta', False, acquired True ? take-sb O', Map.empty), m, S))\).

apply (rule sim-config,intros)
apply (simp add: \(ts' sb' O' sb' R' sb' m\)
flush-all-unti-volatile-nth-update-unused [OF i-bound ts[i-bound i])
using (clarsimp simp add: \(ts' sb' S' sb' m, S\)
apply (clarsimp simp add: \(ts' sb' O' sb' R' sb' \theta' sb' S' sb)\)
using leq
apply (simp add: \(ts' sb\)
using i-bound i-bound
apply (clarsimp simp add: \(ts' sb' O' sb' R' sb' \theta' sb' S' sb\)
dxnot: if-split-asm )
done
ultimately
show ?thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-sops'
valid-dd' load-tmps-fresh' enough-flushs'
valid-program-history' valid' m' sb' S' sb'
by (auto simp del: fun-upd-apply)
next
  case (SBHRMWRDReadOnly cond t a D f ret A L R W)
  then obtain
is sb: is sb = RMW a t (D,f) cond ret A L R W # is sb' and
cond: \(\neg (\text{cond } (\theta sb (t \mapsto m sb a))))\) and
\(O sb' = O sb\) and
\(R sb' = R sb\) and
\(\theta sb' = \theta sb (t \mapsto m sb a)\) and
\(D sb' = D sb\) and
sb: sb = [] and
sb': sb' = [] and
m sb': m sb' = m sb and
S sb': S sb' = S sb
by auto

from safe-RMW-common [OF safe-memop-flush-sb [simplified is sb]]
obtain access-cond: a \(\in O sb \ \vee \ a \in \text{dom } S\) and
rels-cond: \(\forall j < \text{length ts. } i \neq j \longrightarrow \text{released } (ts!j) a \neq \text{Some False}\)
by (auto simp add: S sb)

have valid-own': valid-ownership S sb' ts sb'
proof (intro-locale)
show outstanding-non-volatile-refs-owned-or-read-only S sb' ts sb'
proof –
have non-volatile-owned-or-read-only False $S_{sb}$ $O_{sb}$ []
by simp
from outstanding-non-volatile-ref-owned-or-read-only-nth-update [OF i-bound this]
show ?thesis by (simp add: $ts_{sb}$' $sb$' $sb$ $O_{sb}$' $S_{sb}$')
qed

next
from outstanding-volatile-writes-unowned-by-others-store-buffer
[OF i-bound $ts_{sb}$-i subset-refl]
show outstanding-volatile-writes-unowned-by-others $ts_{sb}$'
by (simp add: $ts_{sb}$' $sb$' $sb$ $O_{sb}$' $S_{sb}$')
next
from read-only-reads-unowned-nth-update [OF i-bound $ts_{sb}$-i, of [] $O_{sb}$]
show read-only-reads-unowned $ts_{sb}$'
by (simp add: $ts_{sb}$' $sb$' $sb$ $O_{sb}$')
next
from ownership-distinct-instructions-read-value-store-buffer-independent
[OF i-bound $ts_{sb}$-i]
show ownership-distinct $ts_{sb}$'
by (simp add: $ts_{sb}$' $sb$' $sb$ $O_{sb}$')
qed

have valid-hist': valid-history program-step $ts_{sb}$'
proof -
from valid-history [OF i-bound $ts_{sb}$-i]
have history-consistent ($\theta_{sb}(t\mapsto m_{sb} a))$ (hd-prog $p_{sb}$ []) [] by simp
from valid-history-nth-update [OF i-bound this]
show ?thesis by (simp add: $ts_{sb}$' $sb$' $sb$ $O_{sb}$' $\theta_{sb}$')
qed

have valid-reads': valid-reads $m_{sb}$ $ts_{sb}$'
proof -
have reads-consistent False $O_{sb}$ $m_{sb}$ [] by simp
from valid-reads-nth-update [OF i-bound this]
show ?thesis by (simp add: $ts_{sb}$' $sb$' $sb$ $O_{sb}$')
qed

have valid-sharing': valid-sharing $S_{sb}$' $ts_{sb}$'
proof (intro-locales)
from outstanding-non-volatile-writes-unshared [OF i-bound $ts_{sb}$-i]
have non-volatile-writes-unshared $S_{sb}$ []
by (simp add: sb)
from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared $S_{sb}$' $ts_{sb}$'
by (simp add: $ts_{sb}$' $sb$' $sb$' $S_{sb}$')
next
have sharing-consistent $S_{sb}$ $O_{sb}$ [] by simp
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis $S_{sb}$' $ts_{sb}$'

538
by (simp add: ts\sb\prime \ O\sb\prime \ s\sb\prime \ sb \ S\sb\prime )

next

from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts\sb\prime-i]]

show read-only-unowned S\sb\prime \ ts_\sb\prime

by (simp add: S\sb\prime \ ts\sb\prime \ O\sb\prime \ s\sb\prime \ sb \ S\sb\prime )

next

from unowned-shared-nth-update [OF i-bound ts\sb\prime-i subset-refl]

show unowned-shared S\sb\prime \ ts\sb\prime \ by (simp add: ts\sb\prime \ sb\prime \ sb \ O\sb\prime \ S\sb\prime)

next

from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound]

show no-outstanding-write-to-read-only-memory S\sb\prime \ ts\sb\prime

by (simp add: S\sb\prime \ ts\sb\prime \ sb \ s\sb\prime)

qed

have tmps-distinct_\prime : tmps-distinct ts\sb\prime

proof (intro-locales)

from load-tmps-distinct [OF i-bound ts\sb\prime-i]

have distinct-load-tmps is\sb\prime

by (auto simp add: is\sb \ split: instr.splits)

from load-tmps-distinct-nth-update [OF i-bound this]

show load-tmps-distinct ts\sb\prime \ by (simp add: ts\sb\prime \ sb\prime \ sb \ O\sb\prime \ is\sb\prime)

next

from read-tmps-distinct [OF i-bound ts\sb\prime-i]

have distinct-read-tmps [ ] by (simp add: ts\sb\prime \ sb\prime \ sb \ O\sb\prime)

from read-tmps-distinct-nth-update [OF i-bound this]

show read-tmps-distinct ts\sb\prime \ by (simp add: ts\sb\prime \ sb\prime \ sb \ O\sb\prime)

next

from load-tmps-read-tmps-distinct [OF i-bound ts\sb\prime-i]

load-tmps-distinct [OF i-bound ts\sb\prime-i]

have load-tmps is\sb\prime \ \cap \ read-tmps [ ] = { }

by (clarsimp)

from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]

show load-tmps-read-tmps-distinct ts\sb\prime \ by (simp add: ts\sb\prime \ sb\prime \ sb \ O\sb\prime)

qed

have valid-sops_\prime : valid-sops ts\sb\prime

proof –

from valid-store-sops [OF i-bound ts\sb\prime-i]

obtain

valid-store-sops_\prime : \ \forall \ sop \ \in \ store-sops \ is\sb\prime \ \mbox{valid-sop sop}

by (auto simp add: is\sb \ sb\prime \ sb \ O\sb\prime)

from valid-sops-nth-update [OF i-bound - valid-store-sops\prime, where sb= []]

show ?thesis by (auto simp add: ts\sb\prime \ sb\prime \ sb \ O\sb\prime)

qed

have valid-dd_\prime : valid-data-dependency ts\sb\prime

proof –

from data-dependency-consistent-instrs [OF i-bound ts\sb\prime-i]
obtain
  dd-is: data-dependency-consistent-instrs (dom \( \vartheta_{sb}' \)) is \( \vartheta_{sb}' \)
by (auto simp add: is_{sb} \( \vartheta_{sb}' \))
from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' \( \cap \bigcup (\text{fst ' write-sops []}) = {} \)
by (auto simp add: write-sops-append)
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show ?thesis by (simp add: ts_{sb}' sb' sb O_{sb}')
qed

have load-tmps-fresh': load-tmps-fresh ts_{sb}'
proof –
from load-tmps-fresh [OF i-bound ts_{sb}-i]
have load-tmps (RMW a t (D,f) cond ret A L R W# is_{sb}') \( \cap \) dom \( \vartheta_{sb} = {} \)
by (auto simp add: is_{sb})
moreover
from load-tmps-distinct [OF i-bound ts_{sb}-i] have t \( \notin \) load-tmps is_{sb}'
by (auto simp add: is_{sb})
ultimately have load-tmps is_{sb}' \( \cap \) dom (\( \vartheta_{sb}(t \mapsto \_ m_{sb} a) \)) = {}
by auto
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}' sb' \( \vartheta_{sb}' \))
qed

from enough-flushs-nth-update [OF i-bound, where sb=[]]
have enough-flushs': enough-flushs ts_{sb}'
by (auto simp add: ts_{sb}' sb' sb)

have valid-program-history': valid-program-history ts_{sb}'
proof –
have causal': causal-program-history is_{sb}' sb'
by (simp add: is_{sb} sb sb')
have last-prog p_{sb} sb' = p_{sb}
by (simp add: sb' sb)
from valid-program-history-nth-update [OF i-bound causal' this]
show ?thesis
by (simp add: ts_{sb}' sb')
qed

from is-sim have is: is = RMW a t (D,f) cond ret A L R W# is_{sb}'
by (simp add: suspends sb is_{sb})
with ts-i
have ts-i: ts!i = (p_{sb}, RMW a t (D,f) cond ret A L R W# is_{sb}', \( \vartheta_{sb},() \)), \( \mathcal{D}, \) acquired True ?take-sb O_{sb}, release ?take-sb (dom \( \mathcal{S}_{sb} \)) \( \mathcal{R}_{sb} \)
by (simp add: suspends sb)

have flush-all-until-volatile-write ts_{sb} m_{sb} a = m_{sb} a
proof –
have \( \forall j < \text{length } ts_{sb}, i \neq j \rightarrow \)

(let \((-,-,-,sb_{j},-,-) = ts_{sb}[j]\)

\(\text{in } a \notin \text{outstanding-refs is-non-volatile-Write}_{sb_{j}} \) (takeWhile (Not \(\circ \) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))

proof –

\{ fix \(j p_{j}\) is\(j\) O\(j\) R\(j\) xs\(j\) sb\(j\) \n assume j-bound: \(j < \text{length } ts_{sb}\) \n assume neq-i-j: \(i \neq j\) \n assume jth: \(ts_{sb}[j] = (p_{j},is_{j},xs_{j},sb_{j},D_{j},O_{j},R_{j})\)

have a \(\notin \text{outstanding-refs is-non-volatile-Write}_{sb_{j}} \) (takeWhile (Not \(\circ \) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))

proof

let ?take-sb\(j\) = (takeWhile (Not \(\circ \) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))

let ?drop-sb\(j\) = (dropWhile (Not \(\circ \) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))

assume a-in: a \(\in\) outstanding-refs is-non-volatile-Write\(\)\(sb_{j}\)

with outstanding-refs-takeWhile [where \(P' = \) Not \(\circ \) is-volatile-Write\(\)\(sb_{j}\)\)]

have a-in': a \(\in\) outstanding-refs is-non-volatile-Write\(\)\(sb_{j}\) \(sb_{j}\) by auto

by auto

with non-volatile-owned-or-read-only-outstanding-non-volatile-writes
\[\text{OF outstanding-non-volatile-refs-owned-or-read-only } [\text{OF j-bound jth}]\]

have j-owns: a \(\in\) O\(j\) \(\cup\) all-acquired sb\(j\)

by auto

from rels-cond [rule-format, OF j-bound [simplified leq] neq-i-j ts-sim [rule-format, OF j-bound] jth

have no-unsharing-release ?take-sb\(j\) (dom (S_{sb})) \(R_{j}\) a \(\neq\) Some False

by (auto simp add: Let-def)

from access-cond

show False

proof

assume a \(\in\) O\(sb\)

with ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i jth]

j-owns

show False

by auto

next

assume a-shared: a \(\in\) dom S

with share-all-until-volatile-write-thread-local [OF ownership-distinct-ts_{sb}
sharing-consis-ts_{sb} j-bound jth j-owns]

have a-dom: a \(\in\) dom (share ?take-sb\(j\) S_{sb})

by (auto simp add: S domIff)

from outstanding-non-volatile-writes-unshared [OF j-bound jth]

have non-volatile-writes-unshared S_{sb} sb_{j}.

with non-volatile-writes-unshared-append [of S_{sb} (takeWhile (Not \(\circ\) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))

(dropWhile (Not \(\circ \) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))]

have unshared-take: non-volatile-writes-unshared S_{sb} (takeWhile (Not \(\circ\) is-volatile-Write\(\)\(sb_{j}\)) \(sb_{j}\))

by clarsimp

541
from release-not-unshared-no-write-take [OF unshared-take no-unsharing

a-dom] a-in

show False by auto

qed

qed

thus \textit{thesis}

by (fastforce simp add: Let-def)

qed

from flush-all-until-volatile-write-buffered-val-conv

[OF - i-bound ts_{sb}-i this]

show \textit{thesis}

by (simp add: sb)

qed

hence m-a: m a = m_{sb} a

by (simp add: m)

from cond have cond': \neg (\vartheta_{sb}(t \mapsto m a))

by (simp add: m-a)

from direct-memop-step.RMWReadOnly [where cond=cond and \vartheta=\vartheta_{sb} and m=m,

OF cond']

have (RMW a t (D, f) cond ret A L R W # is_{sb}',

\vartheta_{sb}', () , D, O_{sb}, R_{sb}, S) \rightarrow

(is_{sb}', \vartheta_{sb}(t \mapsto m a), (), m, False, O_{sb}, Map.empty, S),

from direct-computation.concurrent-step.Memop [OF i-bound' ts-i [simplified sb, simplified] this]

have (ts, m, S) \Rightarrow_d (ts[i := (p_{sb}, i_{sb}', \vartheta_{sb}(t \mapsto m a), () , False, O_{sb}, Map.empty)], m, S).

moreover

have tmps-commute: \vartheta_{sb}(t \mapsto (m_{sb} a)) =

(\vartheta_{sb} |' (dom \vartheta_{sb} - \{t\}))(t \mapsto (m_{sb} a))

apply (rule ext)

apply (auto simp add: restrict-map-def domIff)

done

have (ts_{sb}',m_{sb},S_{sb}') \sim (ts[i := (p_{sb},i_{sb}', \vartheta_{sb}(t \mapsto m a),(), False,O_{sb},Map.empty)],m,S)

apply (rule sim-configintros)

apply (simp add: ts_{sb}'/sb'O_{sb}'R_{sb}' m

flush-all-until-volatile-nth-update-unused [OF i-bound ts_{sb}-i, simplified sb])

using share-all-until-volatile-write-RMW-commute [OF i-bound ts_{sb}-i [simplified is_{sb} sb]]

apply (clarsimp simp add: S ts_{sb}' S_{sb}' is_{sb} O_{sb}' \vartheta_{sb}' sb' sb)

using leq

apply (simp add: ts_{sb}')

using i-bound i-bound' ts-sim

542
apply (clarsimp simp add: Let-def nth-list-update
  \ts_{sb}' \sb' \sb \sigma_{sb}' \mathcal{R}_{sb}' \mathcal{S}_{sb}' \mathcal{D}_{sb}' \text{ex-not m-a}
  \text{split: if-split-asm})
apply (rule tmps-commute)
done
ultimately
show \thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmpps-distinct' valid-sops'
valid-dd' load-tmps-fresh' enough-flushs'
valid-program-history' valid' \msb' \mathcal{S}_{sb}'
by (auto simp del: fun-upd-apply)
next
case (SBHRMWWrite cond t a D f ret A L R W)
then obtain
is_{sb}: \is_{sb} = \text{RMW a t (D,f) cond ret A L R W \\ # is_{sb}' and}
cond: (\text{cond (} \sigma_{sb}(t\rightarrow \msb a) )) \text{ and}
\sigma_{sb}': \\sigma_{sb}' = \sigma_{sb} \cup A - R \text{ and}
\mathcal{R}_{sb}': \\mathcal{R}_{sb}' = \text{Map.empty and}
\mathcal{D}_{sb}': \neg \mathcal{D}_{sb} ' \text{ and}
\theta_{sb}': \theta_{sb}' = \theta_{sb}(t\rightarrow \text{ret (} \msb a (f (\sigma_{sb}(t\rightarrow \msb a)))) \text{) and}
\sb: \sb = [] \text{ and}
\sb': \sb' = [] \text{ and}
\msb': \msb' = \msb(a := f (\sigma_{sb}(t\rightarrow \msb a))) \text{ and}
\mathcal{S}_{sb}': \mathcal{S}_{sb}' = \mathcal{S}_{sb} \oplus W R \ominus A L
by auto

from data-dependency-consistent-instrs [OF i-bound ts_{sb}-i]
  have D-subset: D \subseteq \text{dom } \sigma_{sb}
by (simp add: is_{sb})

from is-sim have is: \is = \text{RMW a t (D,f) cond ret A L R W \\ # is_{sb}'}
by (simp add: suspends sb is_{sb})
with ts-i
have ts-i: ts!i = (p_{sb}, \text{RMW a t (D,f) cond ret A L R W \\ # is_{sb}', } \sigma_{sb}(.), \mathcal{D}, \sigma_{sb}', \mathcal{R}_{sb})
by (simp add: suspends sb)

from safe-RMW-common [OF safe-memop-flush-sb [simplified is_{sb}]]
obtain access-cond: a \in \sigma_{sb} \lor a \in \text{dom } \mathcal{S} \text{ and}
rels-cond: \forall j < \text{length ts. } i \neq j \longrightarrow \text{released (ts!j) a} \neq \text{Some False}
by (auto simp add: \mathcal{S} \sb)

have a-unflushed:
\forall j < \text{length ts}_{sb}. \ i \neq j \longrightarrow
(\text{let } (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = ts_{sb}[j]
\text{ \in a } \notin \text{ outstanding-refs is-non-volatile-Write}_{sb} \text{ (takeWhile (Not o is-volatile-Write}_{sb} \sb))}
proof –

{543}
\[
\text{fix } j \ p_j \ is_j \ O_j \ R_j \ D_j \ x_s_j \ sb_j \\
\text{assume } \text{j-bound: } j < \text{length ts}_{sb} \\
\text{assume } \text{neq-i-j: } i \neq j \\
\text{assume } \text{jth: } ts_{sb}[j] = (p_j,is_j, x_s_j, sb_j, D_j, O_j, R_j) \\
\text{have } a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \\
\text{proof} \\
\text{let } ?\text{take-sb}_j = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \\
\text{let } ?\text{drop-sb}_j = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \\
\text{assume } a\text{-in: } a \in \text{outstanding-refs is-non-volatile-Write}_{sb} ?\text{take-sb}_j \\
\text{with } \text{outstanding-refs-takeWhile } [\text{where } P' = \text{Not } \circ \text{is-volatile-Write}_{sb}] \\
\text{have } a\text{-in'}: a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb}_j \\
\text{by } \text{auto} \\
\text{with } \text{non-volatile-owned-or-read-only-outstanding-non-volatile-writes} \\
[\text{OF outstanding-non-volatile-refs-owned-or-read-only } [\text{OF j-bound jth}]] \\
\text{have } j\text{-owns: } a \in O_j \cup \text{all-acquired sb}_j \\
\text{by } \text{auto} \\
\text{with } \text{ownership-distinct } [\text{OF i-bound j-bound neq-i-j ts}_{sb}-i jth] \\
\text{have } a\text{-not-owns: } a \notin O_{sb} \cup \text{all-acquired sb} \\
\text{by } \text{blast} \\
\text{assume } a\text{-in: } a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \\
(\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \\
\text{with } \text{outstanding-refs-takeWhile } [\text{where } P' = \text{Not } \circ \text{is-volatile-Write}_{sb}] \\
\text{have } a\text{-in'}: a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb}_j \\
\text{by } \text{auto} \\
\text{from } \text{rels-cond } [\text{rule-format, OF j-bound [simplified leq] neq-i-j ts-sim } [\text{rule-format, OF j-bound jth}]] \\
\text{have } j\text{-owns} \\
\text{show } \text{False} \\
\text{proof} \\
\text{assume } a \in O_{sb} \\
\text{with } \text{ownership-distinct } [\text{OF i-bound j-bound neq-i-j ts}_{sb}-i jth] \\
\text{j-owns} \\
\text{show } \text{False} \\
\text{by } \text{auto} \\
\text{next} \\
\text{assume } a\text{-shared: } a \in \text{dom } S \\
\text{with } \text{share-all-until-volatile-write-thread-local } [\text{OF ownership-distinct-ts}_{sb} \\
\text{sharing-consis-ts}_{sb} \text{j-bound jth j-owns}] \\
\text{have } a\text{-dom: } a \in \text{dom } (\text{share } ?\text{take-sb}_j S_{sb}) \\
\text{by } \text{(auto simp add: Let-def)} \\
\text{from } \text{access-cond} \\
\text{show } \text{False} \\
\text{proof} \\
\text{assume } a \in O_{sb} \\
\text{with } \text{ownership-distinct } [\text{OF i-bound j-bound neq-i-j ts}_{sb}-i jth] \\
\text{j-owns} \\
\text{show } \text{False} \\
\text{by } \text{auto} \\
\text{next} \\
\text{assume } a\text{-shared: } a \in \text{dom } S \\
\text{with } \text{share-all-until-volatile-write-thread-local } [\text{OF ownership-distinct-ts}_{sb} \\
\text{sharing-consis-ts}_{sb} \text{j-bound jth j-owns}] \\
\text{have } a\text{-dom: } a \in \text{dom } (\text{share } ?\text{take-sb}_j S_{sb}) \\
\text{by } \text{(auto simp add: S domIff)} \\
\text{from } \text{outstanding-non-volatile-writes-unshared } [\text{OF j-bound jth}] \\
\text{have } \text{non-volatile-writes-unshared } S_{sb} \text{ sb}_j. \\
\text{with } \text{non-volatile-writes-unshared-append } [\text{of } S_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \\
(\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j)] \\
\text{have } \text{unshared-take: } \text{non-volatile-writes-unshared } S_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \]
by clarsimp

  from release-not-unshared-no-write-take [OF unshared-take no-unsharing a-dom] a-in
  show False by auto
  qed
  qed
}

thus ?thesis
  by (fastforce simp add: Let-def)
  qed

  have flush-all-until-volatile-write ts sb m sb a = m sb a
    proof
      from flush-all-until-volatile-write-buffered-val-conv
        [OF - i-bound ts sb - i a-unflushed]
      show ?thesis
        by (simp add: sb)
      qed

      hence m-a: m a = m sb a
        by (simp add: m)
      from cond have cond': cond (\sb(t \mapsto m a))
        by (simp add: m-a)

      from safe-memop-flush-sb [simplified is sb] cond'
        obtain
          L-subset: L \subseteq A and
          A-shared-owned: A \subseteq \dom S \cup \dom O sb and
          R-owned: R \subseteq \dom O sb and
            A-R: A \cap R = {} and
            a-unowned-others-ts:
              \forall j < \length ts. i \neq j \rightarrow (a \notin \owned (ts!j) \cup \dom (\released (ts!j))) and
            A-unowned-by-others-ts:
              \forall j < \length ts. i \neq j \rightarrow (A \cap (\owned (ts!j) \cup \dom (\released (ts!j)))) = {} and
            a-not-ro: a \notin \read-only S
          by cases (auto simp add: sb)

      from a-unowned-others-ts ts-sim leq
      have a-unowned-others:
        \forall j < \length ts sb. i \neq j \rightarrow
          (let (-,-,-,sb j,-,O j,-) = ts sb!j in
          a \notin \acquired True (takeWhile (Not \circ \is-volatile-Write sb) sb j) O j \land
          a \notin \all-shared (takeWhile (Not \circ \is-volatile-Write sb) sb j))
      apply (clarsimp simp add: Let-def)
      subgoal for j
      apply (drule-tac x=j in spec)
      apply (auto simp add: dom-release-takeWhile)
done
done

from A-unowned-by-others-ts ts-sim leq
have A-unowned-by-others:
\[ \forall j < \text{length } ts_{sb} \cdot i \neq j \rightarrow (\text{let } (-, - \cdot sb_{j}, -) = ts_{sb}!j \text{ in } A \cap (\text{acquired } \text{True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}) O_{j} \cup \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}) = \{\})) \]
apply (clarsimp simp add: Let-def)
subgoal for \( j \)
apply (drule-tac \( x=j \) in spec)
apply (force simp add: dom-release-takeWhile)
done
done

have a-not-ro': a \notin \text{read-only } S_{sb}
proof
assume a: a \in \text{read-only } (S_{sb})
from local.read-only-unowned-axioms have read-only-unowned \( S_{sb} \) ts_{sb}.
from in-read-only-share-all-until-volatile-write' [OF ownership-distinct-ts sb]
sharing-consis-ts_{sb}
(read-only-unowned \( S_{sb} \) ts_{sb}) i-bound ts_{sb}-i a-unowned-others, simplified sb, simplified,
OF \( a \)
have a \in \text{read-only } (S)
by (simp add: \( S \))
with a-not-ro show False by simp
qed

{ 
fix \( j \)
fix \( p_{j} \) is_{sbj} O_{j} R_{sbj} D_{sbj} \emptyset_{j} sb_{j}
assume j-bound: \( j < \text{length } ts_{sb} \)
assume ts_{sb}-j: ts_{sb}!j = (p_{j}, \text{is}_{sbj}, \emptyset_{j}, sb_{j}, D_{sbj}, O_{j}, R_{j})
assume neq-i-j: i \neq j
have a \notin \text{unforwarded-non-volatile-reads } (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}) \{\}
proof
let ?take-sb_{j} = \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}
let ?drop-sb_{j} = \text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_{j}
assume a-in: a \in \text{unforwarded-non-volatile-reads } ?\text{drop-sb}_{j} \{\}

from a-unowned-others [rule-format, OF - neq-i-j] ts_{sb}-j j-bound
obtain a-unacq-take: a \notin \text{acquired } \text{True } ?\text{take-sb}_{j} O_{j} \text{ and } a\text{-not-shared: } a \notin \text{all-shared } ?\text{take-sb}_{j}
by auto

note nvo-j = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound ts_{sb}-j]
from non-volatile-owned-or-read-only-drop [OF nvo-j]

546
have nvo-drop-j: non-volatile-owned-or-read-only True (share ?take-sbj \(S_{sb}\))
(acquired True ?take-sb \(O_j\)) ?drop-sb_j .

note consis-j = sharing-consis [OF j-bound ts\(sb\)-j]
with sharing-consistent-append [of \(S_{sb}\) \(O_j\) ?take-sb ?drop-sb]
obtain consis-take-j: sharing-consistent \(S_{sb}\) \(O_j\) ?take-sb_j and
consis-drop-j: sharing-consistent (share ?take-sb_j \(S_{sb}\))
(acquired True ?take-sb_j \(O_j\)) ?drop-sb_j
by auto

from in-unforwarded-non-volatile-reads-non-volatile-Read\(sb\) [OF a-in]
have a-in": a ∈ outstanding-refs is-non-volatile-Read\(sb\) ?drop-sb_j.

note reads-consis-j = valid-reads [OF j-bound ts\(sb\)-j]
from reads-consistent-drop [OF this]
have reads-consis-drop-j:
reads-consistent True (acquired True ?take-sb_j \(O_j\)) (flush ?take-sb_j m\(sb\)) ?drop-sb_j.

from read-only-share-all-shared [of a ?take-sb_j \(S_{sb}\)] a-not-ro' a-not-shared
have a-not-ro-j: a \∉ read-only (share ?take-sb_j \(S_{sb}\))
by auto

from ts-sim [rule-format, OF j-bound] ts\(sb\)-j j-bound
obtain suspends\(j\) is\(j\) \(D_j\) where
suspends\(j\): suspends\(j\) = ?drop-sb_j and
is\(j\): instrs suspends\(j\) @ is\(sb\)_j = is\(j\) @ prog-instrs suspends\(j\) and
\(D_j\): \(D_{sbj}\) = (\(D_j\) ∨ outstanding-refs is-volatile-Write\(sb\) \(sb\)_j ≠ \{\}) and
\(ts_j\): ts\(j\) = (hd-prog \(p_j\) suspends\(j\), is\(j\),
\(\theta_j\) | (dom \(\theta_j\) − read-tmps suspends\(j\)),),
\(D_j\), acquired True ?take-sb \(O_j\), release ?take-sb_j (dom \(S_{sb}\) \(R_j\))
by (auto simp: Let-def)

from ts\(j\) neq-i-j j-bound
have ts\(j\)': ts\(j\) = (hd-prog \(p_j\) suspends\(j\), is\(j\),
\(\theta_j\) | (dom \(\theta_j\) − read-tmps suspends\(j\)),),
\(D_j\), acquired True ?take-sb \(O_j\), release ?take-sb_j (dom \(S_{sb}\) \(R_j\))
by auto

from valid-last-prog [OF j-bound ts\(sb\)-j] have last-prog: last-prog \(p_j\) \(sb\)_j = \(p_j\).

from j-bound i-bound' leq have j-bound-ts\(j\): j < length ?ts'
by simp

from read-only-read-acquired-unforwarded-acquire-witness [OF nvo-drop-j consis-drop-j
a-not-ro-j a-unacq-take a-in]
have False

547
proof
  assume \exists \text{sop a'} v \ ys \ zs \ A \ L \ R \ W.
  \?drop-sb \ j = ys \ @ \ \text{Write}_{st} \ \text{True} \ \text{a'} \ \text{sop} \ v \ A \\ L \ R \ W \# \ zs \ \& \ a \in A \ \&
  \ a \notin \text{outstanding-refs} \ \text{is-Write}_{st} \ ys \ \& \ a' \neq a
  \text{with} \ \text{suspends}_{st \ j}
  \text{obtain} \ a' \ \text{sop}' \ v' \ ys \ zs' \ A' \ L' \ R' \ W' \ \text{where}
  \text{split-suspend}_{st} = ys \ @ \ \text{Write}_{st} \ \text{True} \ \text{a'} \ \text{sop}' \ v' \ A' \ L' \ R' \ W' \# \ zs' \ \text{(is} \ \text{suspend}_{st \ j} = \? \text{suspend}) \ \text{and}
  a-A'; \ a \in A' \ \text{and}
  \text{no-write:} \ a \notin \text{outstanding-refs} \ \text{is-Write}_{st} \ (ys \ @ \ \text{Write}_{st} \ \text{True} \ \text{a'} \ \text{sop}' \ v' \ A' \ L' \ R' \ W')
  \text{by (auto simp add: outstanding-refs-append)}

from \ \text{last-prog}
  have \ \text{lp}: \ \text{last-prog} \ p \ suspends_{st \ j} = p_j
    apply
    apply (rule \ \text{last-prog-same-append} \ [\\text{where sb=?take-sb}])
    apply (simp only: \ \text{split-suspend}_{st \ j} \ \text{suspend} \ [\text{symmetric}] \ \text{suspend}_{st \ j})
    apply simp
    done

from \ \text{sharing-consis} \ \text{[OF j-bound ts}_{st \ j}]
  have \ \text{sharing-consis-j}: \ \text{sharing-consistent} \ \mathcal{S}_{st} \ O_j \ sb_j.
  then \ have \ A'-R': \ A' \cap R' = \{\}
    \text{by (simp add: sharing-consistent-append} \ [\\text{of} \ - \ ? \text{take-sb} \ ? \text{drop-sb}, \ \text{simplified}]
  \ \text{suspend}_{st \ j} \ \text{[symmetric]} \ \text{split-suspend}_{st \ j} \ \text{sharing-consistent-append})

from \ \text{valid-program-history} \ \text{[OF j-bound ts}_{st \ j}]
  have \ \text{causal-program-history} \ is_{stb \ j} \ sb_j.
  then \ have \ \text{cph}: \ \text{causal-program-history} \ is_{stb \ j} ? \ \text{suspend}
    apply
    apply (rule \ \text{causal-program-history-suffix} \ [\\text{where sb=?take-sb}])
    apply (simp only: \ \text{split-suspend}_{st \ j} \ \text{suspend} \ [\text{symmetric}] \ \text{suspend}_{st \ j})
    apply (simp add: \ \text{split-suspend}_{st \ j})
    done

from \ \text{valid-reads} \ \text{[OF j-bound ts}_{st \ j}]
  have \ \text{reads-consis-j}: \ \text{reads-consistent} \ \text{False} \ O_j \ m_{stb} \ sb_j.

from \ \text{reads-consistent-flush-all-until-volatile-write} \ \text{[OF (valid-ownership-and-sharing}
  \mathcal{S}_{st} \ \text{ts}_{st} \ j) \ \text{j-bound ts}_{st \ j} \ \text{this}]
  have \ \text{reads-consis-m-j}: \ \text{reads-consistent} \ \text{True} \ \text{(acquired True} \ ? \text{take-sb} \ O_j \ m \ \text{suspend}_{st \ j} \ \text{by (simp add: m suspend}_{st \ j})

from \ \text{outstanding-non-write-non-vol-reads-drop-disj} \ \text{[OF i-bound j-bound neq-i-j ts}_{st \ i} \ \text{ts}_{st \ j}]\n  have \ \text{outstanding-refs is-Write}_{st} \ ? \ \text{drop-sb} \ \cap \ \text{outstanding-refs} \ \text{is-non-volatile-Read}_{st}
  \ \text{suspend}_{st \ j} = \{\}
  \text{by (simp add: suspend}_{st \ j})
  from \ \text{reads-consistent-flush-independent} \ \text{[OF this \ \text{reads-consis-m-j]}

548
have reads-consis-flush-suspend: reads-consistent True (acquired True ?take-sb \( j \) \( \mathcal{O} \)) (flush ?drop-sb m) suspends\( j \).

hence reads-consis-ys: reads-consistent True (acquired True ?take-sb \( j \) \( \mathcal{O} \)) (flush ?drop-sb m) (ys@\( Write_{sb} \) True \( a' \) \( sop' \) \( v' \) \( A' \) \( L' \) \( R' \) \( W' \))
by (simp add: split-suspends\( j \) reads-consistent-append)

from valid-write-sops [OF j-bound ts_{sb-j}]
have \( \forall \) sop\( \in \) write-sops (?take-sb \( j \) ?suspends). valid-sop sop
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
then obtain valid-sops-take: \( \forall \) sop\( \in \) write-sops ?take-sb\( j \). valid-sop sop and
valid-sops-drop: \( \forall \) sop\( \in \) write-sops (ys@\( Write_{sb} \) True \( a' \) \( sop' \) \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)). valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound ts_{sb-j}]
have distinct-read-tmps (?take-sb \( j \) @suspends\( j \))
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
then obtain read-tmps-take-drop: read-tmps ?take-sb \( j \) \( \cap \) read-tmps suspends\( j \) = \{\}
and
distinct-read-tmps-drop: distinct-read-tmps suspends\( j \)
apply (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound ts_{sb-j}]
have h-consis:
history-consistent \( \theta \) \( j \) (hd-prog \( p \) \( j \) (?take-sb \( j \) @suspends\( j \))) (?take-sb \( j \) @suspends\( j \))
apply (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog \( p \) \( j \) \( sb \) \( j \)) ?take-sb\( j \) = (hd-prog \( p \) \( j \) suspends\( j \))
proof –
from last-prog have last-prog \( p \) \( j \) (?take-sb\( j \) ?drop-sb\( j \)) = \( p \)
by simp
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog \( p \) \( j \) suspends\( j \)) ?take-sb\( j \) = hd-prog \( p \) \( j \) suspends\( j \)
by (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
moreover
have last-prog (hd-prog \( p \) \( j \) (?take-sb\( j \) @ suspends\( j \))) ?take-sb\( j \) =
last-prog (hd-prog \( p \) \( j \) suspends\( j \)) ?take-sb\( j \)
apply (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
by (rule last-prog-hd-prog-append)
ultimately show \( \theta \)thesis
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
qed

549
from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop]
  h-consis] last-prog-hd-prog

have hist-consis': history-consistent \( \theta_j \) (hd-prog \( p_j \) suspends\( s_j \)) suspends\( s_j \)
  by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-\( j \)]
have no-vol-read: outstanding-refs is-volatile-Read\( sb \)
  (ys@[Write\( sb \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)]) = {}
  by (auto simp add: outstanding-refs-append suspends\( j \) [symmetric]
split-suspends\( j \))
from flush-store-buffer-append [where \( sb=ys@[Write\( sb \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)]\]
and \( sb'=zs' \), simplified,
  OF j-bound-ts\( i_j \) [simplified split-suspends\( j \)] cph [simplified suspends\( j \)]
  ts'\( j \) [simplified split-suspends\( j \)] refl lp [simplified split-suspends\( j \)] reads-consis-ys
  hist-consis' [simplified split-suspends\( j \)] valid-sops-drop
  distinct-read-tmps-drop [simplified split-suspends\( j \)] no-volatile-Read\( sb \)-volatile-reads-consistent [OF no-vol-read], where
  \( S=\text{share } ?\text{drop-sb } S \)

obtain is\( j' \) R\( j' \) where
  is\( j' \): instrs zs'@[Write\( sb \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)]\]
  steps-ys: (?ts' flush \( ?\text{drop-sb } m \), share \( ?\text{drop-sb } S \) ) \( \Rightarrow \) \( d^* \)
  (?ts'\( j\) := (last-prog
    (hd-prog \( p_j \) (Write\( sb \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)\# zs')), \( ys@[Write\( sb \)
    True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)`)\]
    \( is\( j' \),
    \( \theta_j \mid ^{i} \) (dom \( \theta_j \) - read-tmps zs'),
    \( () \), True, acquired True ys (acquired True \( ?\text{take-sb } O_j \)) \( \cup \) \( A' \) - \( R' \), R\( j' \)],
  \( \text{flush } (ys@[Write\( sb \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)`)\]
    \( \text{share } (ys@[Write\( sb \) True \( a' \) sop' \( v' \) \( A' \) \( L' \) \( R' \) \( W' \)`)\]
    \( \text{share } ?\text{drop-sb } S \))\]
  \( \text{(is } \sim \text{-}) \Rightarrow d^* \) (?ts-ys,?m-ys,?shared-ys)
  by (auto simp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: \( i < \text{length } \text{?ts-ys} \)
  by auto
from i-bound' neq-i-j
have ts-ys-i: ?ts-ys!i = (p\( sb \), is\( sb \), \( \theta sb \),()),
  \( D_{sb} \), acquired True sb \( O_{sb} \), release sb (dom \( S sb \) ) R\( sb \)
  by simp
note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]
from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]

have safe: safe-delayed (?ts-ys,?m-ys,?shared-ys).

from flush-unchanged-addresses [OF no-write]

have flush (ys @ [Write sb True a' sop' v' A' L' R' W']) m a = m a.

with safe-delayedE [OF safe i-bound-ys ts-ys-i]

have a-unowned:

∀ j < length ?ts-ys. i≠j → (let (Cj) = map owned ?ts-ys\x in a ∉ Cj)

apply cases

apply (auto simp add: Let-def is sb)

done

from a-A' a-unowned [rule-format, of j] neq-i-j j-bound leq A'-R'

show False

by (auto simp add: Let-def)

next

assume ∃ A L R W ys zs. ?drop-sb_j = ys @ Ghost_{sb} A L R W# zs ∧ a ∈ A ∧ a ∉ outstanding-refs is-Write_{sb} ys

with suspends_j

obtain ys zs' A' L' R' W' where

split-suspends_j: suspends_j = ys @ Ghost_{sb} A' L' R' W'# zs' (is suspends_j=?suspends)

and a-A': a ∈ A' and

no-write: a ∉ outstanding-refs is-Write_{sb} (ys @ [Ghost_{sb} A' L' R' W'])

by (auto simp add: outstanding-refs-append)

from last-prog

have lp: last-prog p_j suspends_j = p_j

apply −

apply (rule last-prog-same-append [where sb=?take-sb_j])

apply (simp only: split-suspends_j [symmetric] suspends_j)

apply simp

done

from sharing-consis [OF j-bound ts_{sb}-j]

have sharing-consis-j: sharing-consistent S_{sb} C_j sb_j.

then have A'-R': A' ∩ R' = {}

by (simp add: sharing-consistent-append [of - - ?take-sb_j ?drop-sb_j, simplified]

suspends_j [symmetric] split-suspends_j sharing-consistent-append)

from valid-program-history [OF j-bound ts_{sb}-j]

have causal-program-history is_{sbj} sb_j.

then have cph: causal-program-history is_{sbj} ?suspends

apply −

apply (rule causal-program-history-suffix [where sb=?take-sb_j] )

apply (simp only: split-suspends_j [symmetric] suspends_j)

apply (simp add: split-suspends_j)

done
from valid-reads [OF j-bound ts_{sb-j}]
have reads-consis-j: reads-consistent False \(O_j\) m_{sb} sb_{j}.

from reads-consistent-flush-all-until-volatile-write [OF \langle valid-ownership-and-sharing \(S_{sb}\) ts_{sb}\rangle]
\(j\)-bound ts_{sb-j} this]
have reads-consis-m-j: reads-consistent True (acquired True \(?\text{take-sb}_{j} O_j\)) m suspends_{j}
by (simp add: m suspends_{j})

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound neq-i-j ts_{sb-i} ts_{sb-j}]
have outstanding-refs is-Write_{sb} ?drop-sb \(\cap\) outstanding-refs is-non-volatile-Read_{sb}
suspends_{j} = \{\}
by (simp add: suspends_{j})
from reads-consistent-flush-independent [OF this reads-consis-m-j]
have reads-consis-flush-suspend: reads-consistent True (acquired True \(?\text{take-sb}_{j} O_j\))
(flush \(?\text{drop-sb}\) m) suspends_{j}.

hence reads-consis-ys: reads-consistent True (acquired True \(?\text{take-sb}_{j} O_j\))
(flush \(?\text{drop-sb}\) m) \((ys@[Ghost_{sb} A' L' R' W'])\)
by (simp add: split-suspends_{j} reads-consistent-append)

from valid-write-sops [OF j-bound ts_{sb-j}]
have \(\forall\) sop\in write-sops (?\text{take-sb}_{j}@?\text{suspends}). valid-sop sop
by (simp add: split-suspends_{j} [symmetric] suspends_{j})
then obtain valid-sops-take: \(\forall\) sop\in write-sops ?\text{take-sb}_{j}. valid-sop sop and
valid-sops-drop: \(\forall\) sop\in write-sops (ys@[Ghost_{sb} A' L' R' W']). valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound ts_{sb-j}]
have distinct-read-tmps (?\text{take-sb}_{j}@suspends_{j})
by (simp add: split-suspends_{j} [symmetric] suspends_{j})
then obtain read-tmps-take-drop: read-tmps ?\text{take-sb}_{j} \(\cap\) read-tmps suspends_{j} = \{\} and
distinct-read-tmps-drop: distinct-read-tmps suspends_{j}
apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound ts_{sb-j}]
have h-consis:
history-consistent \(\emptyset_j\) (hd-prog p_{j} (?\text{take-sb}_{j}@suspends_{j})) (?\text{take-sb}_{j}@suspends_{j})
apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p_{j} sb_{j}) ?\text{take-sb}_{j} = (hd-prog p_{j} suspends_{j})
proof --
from last-prog have last-prog \( p_j \) (?take-sb\(_j@\)drop-sb\(_j\)) = \( p_j \)
by simp
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog \( p_j \) suspends\(_j\)) ?take-sb\(_j@\)drop-sb\(_j\) = \( p_j \) suspends\(_j\)
by (simp only: split-suspends\(_j \) [symmetric] suspends\(_j\))
moreover
have last-prog (hd-prog \( p_j \) (?take-sb\(_j@\)suspends\(_j\))) ?take-sb\(_j@\)drop-sb\(_j\) = last-prog (hd-prog \( p_j \) suspends\(_j\)) ?take-sb\(_j@\)drop-sb\(_j\)
apply (simp only: split-suspends\(_j \) [symmetric] suspends\(_j\))
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends\(_j \) [symmetric] suspends\(_j\))
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog
have hist-consis': history-consistent \( \theta_j \) (hd-prog \( p_j \) suspends\(_j\)) suspends\(_j\)
by (simp add: split-suspends\(_j \) [symmetric] suspends\(_j\))
from reads-consistent-drop-volatile-writes-no-volatile-reads [OF reads-consis-j]
have no-vol-read: outstanding-refs is-volatile-Read\(_{ab}\)
(ys@[Ghost\(_{ab} A L' R' W'\)]) = {}
by (auto simp add: outstanding-refs-append suspends\(_j \) [symmetric] split-suspends\(_j\))

have acq-simp:
    acquired True (ys @[Ghost\(_{ab} A L' R' W'\)])
    (acquired True \( ?\)take-sb\(_j@\)O\(_j\)\)) =
    acquired True ys (acquired True \( ?\)take-sb\(_j@\)O\(_j\)\) \( \cup \) \( A' - R' \)
by (simp add: acquired-append)

from flush-store-buffer-append [where sb=ys@[Ghost\(_{ab} A L' R' W'\)] and sb'=zs', simplified,
OF j-bound-ts' is\(_j\) [simplified split-suspends\(_j\)] cph [simplified suspends\(_j\)]
ts'\(_j\) [simplified split-suspends\(_j\)] refl lp [simplified split-suspends\(_j\)] reads-consis-ys
hist-consis' [simplified split-suspends\(_j\)] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends\(_j\)]
no-volatile-Read\(_{ab}\)-volatile-reads-consistent [OF no-vol-read], where
\( S=\text{share} ?\)drop-sb \( S \)

obtain is\(_j\)' \( R_j' \), where
is\(_j\)': instrs \( zs' \) @ is\(_{abj}\) = is\(_j\)' @ prog-instrs zs' and
steps-ys: (?ts', flush ?drop-sb m, share ?drop-sb \( S \)) \( \Rightarrow \) \( d' \)
(?ts'[j:=](last-prog
(hd-prog \( p_j \) (Ghost\(_{ab} A L' R' W'\)# zs')) (ys@[Ghost\(_{ab} A L' R' W'\)]),
is\(_j\)' ,
\( \theta_j |' \) (dom \( \theta_j \) - read-tmps zs'),
(),
\( \mathcal{D}_j \) \( \lor \) outstanding-refs is-volatile-Write\(_{ab}\) ys \( \neq \) \{}, acquired True ys
(acquired True \( ?\)take-sb\(_j@\)O\(_j\)\) \( \cup \) \( A' - R', R_j' \)\],

553
flush \((\text{ys}@[\text{Ghost}_{sb} A' L' R' W'])\) (flush \(?\text{drop-sb} m\)),
share \((\text{ys}@[\text{Ghost}_{sb} A' L' R' W'])\) (share \(?\text{drop-sb} S\))
\((\text{is} \ (-,-,-) \Rightarrow \text{flush}^* (\text{ys}@[\text{Ghost}_{sb} A' L' R' W']) (\text{share} \ ?\text{drop-sb} S))\)
bym (auto simp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: \(i < \text{length} \ ?\text{ts-ys}\)
by auto

from i-bound' neq-i-j
have ts-ys-i: \(?\text{ts-ys}!i = (p_{sb}, \text{is}_{sb}, \emptyset_{sb},())\),
\(D_{sb}, \text{acquired} \text{True} sb O_{sb}, \text{release} sb (\text{dom} S_{sb}) R_{sb}\)
by simp
note conflict-computation = rtranclp-trans \([\text{OF steps-flush-sb steps-ys}]\)

from safe-reach-safe-rtrancl \([\text{OF safe-reach conflict-computation}]\)
have safe: \(\text{safe-delayed} (\?\text{ts-ys},\?\text{m-ys},\?\text{shared-ys}).\)

from flush-unchanged-addresses \([\text{OF no-write}]\)
have flush \((\text{ys} @ [\text{Ghost}_{sb} A' L' R' W'])\) \(m a = m a\).

with safe-delayedE \([\text{OF safe i-bound-ys ts-ys-i, simplified is}_{sb}]\) cond'
have a-unowned:
\(\forall j < \text{length} \ ?\text{ts-ys}. \ i \neq j \implies (\text{let} (O_j) = \text{map owned} ?\text{ts-ys}!j \text{ in} a \notin O_j)\)
apply cases
apply (auto simp add: Let-def is_{sb} sb)
done
from a-A' a-unowned \([\text{rule-format, of} j] \ \text{neq-i-j} \ j\)-bound leq A' R'

show False
by (auto simp add: Let-def)
qed
then show False
by simp
qed

note a-notin-unforwarded-non-volatile-reads-drop = this

have A-unused-by-others:
\(\forall j < \text{length} \ (\text{map} O_{sb} ts_{sb}). \ i \neq j \implies\)
(let \((O_j, sb_j) = \text{map} O_{sb} ts_{sb}! j\)
in \(A \cap (O_j \cup \text{outstanding-refs} \text{is-volatile-Write}_{sb} sb_j) = \{\}\))
proof –
{ fix \(j \ \text{C}_{j} \ \text{sb}_j\)
assume j-bound: \(j < \text{length} \ (\text{map} \text{owned} \ ts_{sb})\)
assume neq-i-j: \(i \neq j\)
assume ts_{sb}-j: \((\text{map} O_{sb} ts_{sb})!j = (O_j,sb_j)\)
assume conflict: \(A \cap (O_j \cup \text{outstanding-refs} \text{is-volatile-Write}_{sb} sb_j) \neq \{\}\)

554
have False
proof
  from j-bound leq
  have j-bound': j < length (map owned ts)
    by auto
  from j-bound have j-bound'': j < length ts
    by auto
  from j-bound' have j-bound''': j < length ts
    by simp
from conflict obtain a' where
  a-in: a' ∈ A and
    conflict: a' ∈ O_j ∨ a' ∈ outstanding-refs is-volatile-Write_{sb_j}
  by auto
  with A-unowned-by-others [rule-format, OF - neq-i-j] j-bound ts
  have A-unshared-j: A ∩ all-shared (takeWhile (Not ◦ is-volatile-Write_{sb_j}) sb_j) = {}
    by (auto simp add: Let-def)
from conflict
show ?thesis
proof
  assume a' ∈ O_j
  from all-shared-acquired-in [OF this] A-unshared-j a-in
    conflict: a' ∈ acquired True (takeWhile (Not ◦ is-volatile-Write_{sb_j}) sb_j) O_j
    by auto
  with A-unowned-by-others [rule-format, OF - neq-i-j] j-bound ts a-in
    show False by auto
next
  assume a-in-j: a' ∈ outstanding-refs is-volatile-Write_{sb_j}
  let ?take-sb_j = (takeWhile (Not ◦ is-volatile-Write_{sb_j}) sb_j)
  let ?drop-sb_j = (dropWhile (Not ◦ is-volatile-Write_{sb_j}) sb_j)
  from ts-sim [rule-format, OF] j-bound' j-bound''
    obtain p_j suspends_j is_{sb_j} D_{sb_j} D_j θ_{sb_j} is_j where
      ts_{sb_j}: ts_{sb} ! j = (p_j,is_{sb_j}, θ_{sb_j}, D_{sb_j},O_j,R_j) and
      suspends_j: suspends_j = ?drop-sb_j and
      D_j: D_{sb_j} = (D_j ∨ outstanding-refs is-volatile-Write_{sb_j} sb_j ≠ {}) and
      is_j: instrs suspends_j @ is_{sb_j} = is_j @ prog-intrs suspends_j and
      ts_j: ts_j = (hd-prog p_j suspends_j, is_j, θ_{sb_j} | (dom θ_{sb_j} − read-tmps suspends_j),(), D_j, acquired True ?take-sb_j O_j,release
      ?take-sb_j (dom S_{sb}) R_j)
      apply (cases ts_{sb_j})
apply (force simp add: Let-def)
done
have $a' \in \text{outstanding-refs is-volatile-Write}_s \text{ suspends}_j$

proof

from a-in-j
have $a' \in \text{outstanding-refs is-volatile-Write}_s (\text{?take-sb}_j @ \text{?drop-sb}_j)$
by simp
thus $?\text{thesis}$
apply (simp only: outstanding-refs-append suspends$_j$
apply (auto simp add: outstanding-refs-conv dest: set-takeWhileD)
done
qed

from split-volatile-Write$_s$-in-outstanding-refs [OF this]
obtain sop $v'$ ys zs A' L' R' W' where
split-suspend$_j$: suspends$_j = ys @ \text{Write}_s \text{True} a' s o p' v' A' L' R' W'$# zs (is suspens$_j$
= $?\text{suspends})
by blast

from valid-program-history [OF j-bound'' ts$_{sb}$-j]
have causal-program-history is$_{sbj}$ sb$_j$.
then have cph: causal-program-history is$_{sbj}$ $?\text{suspends}$
apply –
apply (rule causal-program-history-suffix [where sb=?take-sb$_j$ ] )
apply (simp only: split-suspend$_j$ [symmetric] suspends$_j$)
apply (simp add: split-suspend$_j$)
done

from valid-last-prog [OF j-bound'' ts$_{sb}$-j] have last-prog: last-prog p$_j$ sb$_j$ = p$_j$
then
have lp: last-prog p$_j$ $?\text{suspends}$ = p$_j$
apply –
apply (rule last-prog-same-append [where sb=?take-sb$_j$])
apply (simp only: split-suspend$_j$ [symmetric] suspends$_j$)
apply simp
done

from valid-reads [OF j-bound'' ts$_{sb}$-j]
have reads-consis: reads-consistent False $\mathcal{O}_j \text{m} \text{sb} \text{sb}_j$.
from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing
$\mathcal{S}_{sb}$ ts$_{sb}$''
j-bound''
ts$_{sb}$-j this]
have reads-consis-m-j: reads-consistent True (acquired True ?take-sb$_j$ $\mathcal{O}_j$) m suspends$_j$
by (simp add: m suspends$_j$)

from outstanding-non-volatile-refs-owned-or-read-only [OF j-bound'' ts$_{sb}$-j]
have nvo-j: non-volatile-owned-or-read-only False $\mathcal{S}_{sb} \text{O}_j \text{sb}_j$.
with non-volatile-owned-or-read-only-append [of False $\mathcal{S}_{sb} \text{O}_j ?$take-sb$_j$ ?drop-sb$_j$
have nvo-take-j: non-volatile-owned-or-read-only False $\mathcal{S}_{sb} \text{O}_j ?$take-sb$_j$
by auto

from a-unowned-others [rule-format, OF - neq-i-j] ts_{sb-j} j-bound
have a-not-acq: a /∈ acquired True ?take-sb_j \mathcal{O}_j
by auto

from a-notin-unforwarded-non-volatile-reads-drop[OF j-bound"] ts_{sb-j} neq-i-j
have a-notin-unforwarded-reads: a /∈ unforwarded-non-volatile-reads suspends_j {} 
by (simp add: suspends_j)

let \?ma=(m(a := f (\_sb(t→m a))))

from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads [where W={}
and m’=?ma,simplified, OF - subset-refl reads-consis-m-j]
anotin-unforwarded-reads
have reads-consis-ma-j: 
reads-consistent True (acquired True ?take-sb_j \mathcal{O}_j) ?ma suspends_j
by auto

from reads-consis-ma-j
have reads-consis-ys: reads-consistent True (acquired True ?take-sb_j \mathcal{O}_j) ?ma (ys)
by (simp add: split-suspends_j reads-consistent-append)

from direct-memop-step.RMWWrite [where cond=cond and \varnothing=\_sb and m=m, OF cond’]
have (RMW a t (D, f) cond ret A L R W# is_{sb’}, \_sb, (\_sb(t → ret (m a)) (f (\_sb(t → m a)))), (), ?ma, False, \_sb \cup A - R, Map.empty,\_W \cup R \oplus A L).
from direct-computation.concurrent-step.Memop [OF i-bound’ ts-i this]
have step-a: (ts, m, S) ⇒_d
  (ts[i := (p_{sb}, is_{sb’}, \_sb(t → ret (m a)) (f (\_sb(t → m a)))), (), False, \_sb \cup A - R,Map.empty]),
  (?ma,\_W \cup R \oplus A L)
(is - ⇒_d (?ts-a, -, ?shared-a)).

from ts_j neq-i-j j-bound

have ts-a-j: ?ts-alj = (hd-prog p_j suspends_j, is_j, 
\_sbj |' (dom \_sbj - read-tmps suspends_j,()), D_j, acquired True ?take-sb_j \mathcal{O}_j,release ?take-sb_j (dom (\_sbj)) \mathcal{R}_j)
by auto

from valid-write-sops [OF j-bound” ts_{sb-j}]
have \forall sop∈write-sops (?take-sb_j@?suspends). valid-sop sop
by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain valid-sops-take: \forall sop∈write-sops ?take-sb_j. valid-sop sop and valid-sops-drop: \forall sop∈write-sops (ys). valid-sop sop
apply (simp only: write-sops-append)
apply auto

557
from read-tmps-distinct [OF j-bound″ ts_{sb}⁻j]

have distinct-read-tmps (?take-sb_j@suspends_j)
by (simp add: split-suspends_j [symmetric] suspends_j)

then obtain
read-tmps-take-drop: read-tmps ?take-sb_j ∩ read-tmps suspends_j = {} and
distinct-read-tmps-drop: distinct-read-tmps suspends_j

apply (simp only: split-suspends_j [symmetric] suspends_j)
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound″ ts_{sb}⁻j]

have h-consis:
  history-consistent θ_{sbj} (hd-prog p_j (?take-sb_j@suspends_j)) (?take-sb_j@suspends_j)

apply (simp only: split-suspends_j [symmetric] suspends_j)
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
proof –
from last-prog have last-prog p_j (?take-sb_j@?drop-sb_j) = p_j
by simp
from last-prog-hd-prog-append’ [OF h-consis] this
have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
by (simp only: split-suspends_j [symmetric] suspends_j)
moreover
have last-prog (hd-prog p_j (?take-sb_j @ suspends_j)) ?take-sb_j =
last-prog (hd-prog p_j suspends_j) ?take-sb_j

apply (simp only: split-suspends_j [symmetric] suspends_j)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends_j [symmetric] suspends_j)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
  h-consis] last-prog-hd-prog
have hist-consis: history-consistent θ_{sbj} (hd-prog p_j suspends_j) suspends_j
by (simp add: split-suspends_j [symmetric] suspends_j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read_{sb} (ys) = {}
by (auto simp add: outstanding-refs-append suspends_j [symmetric]
split-suspends_j)
from j-bound’ have j-bound-ts-a: j < length ?ts-a by auto

from flush-store-buffer-append [where sb=ys and sb’=Write_{sb} True a’ sop’ v’ A’ L’
R’ W’#zs, simplified,
OF j-bound-ts-a is_j [simplified split-suspends_j] cph [simplified suspends_j]

558
ts-a-j [simplified split-suspends], refl lp [simplified split-suspends], reads-consis-ys
hist-consis'[simplified split-suspends], valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends]

no-volatile-Read_{sb}-volatile-reads-consistent [OF no-vol-read], where
S=?shared-a]

obtain is_{j}' R_{j}' where
is_{j}': Write True a' sop' A' L' R' W'# instrs zs @ is_{sbj} = is_{j}' @ prog-instrs zs and
steps-ys: (?ts-a, ?ma, ?shared-a) \Rightarrow_d?

(?ts-a[j]:=(last-prog (hd-prog p j zs) ys,

(is_{j}',

\theta_{sbj} | (dom \theta_{sbj} - read-tmps zs),

(), D_{j} \cup \text{outstanding-refs is-volatile-Write}_{sb} ys \neq \{\}, acquired True

ys (acquired True ?take-sb_{j} C_{j},R_{j}'),

flush ys (?ma), share ys (?shared-a))

(is _(_,_,_) \Rightarrow_d? (?ts-ys,?m-ys,?shared-ys))

by (auto simp add: acquired-append)

from cph
have causal-program-history is_{sbj} ((ys @ [Write_{sb} True a' sop' v' A' L' R' W']) @ zs)

by simp
from causal-program-history-suffix [OF this]

have cph': causal-program-history is_{sbj} zs.
interpret causal_{j}: causal-program-history is_{sbj} zs by (rule cph')

from causal_{j}.causal-program-history [of []], simplified, OF refl] is_{j}'

obtain is_{j}''

where is_{j}': is_{j}'' = Write True a' sop' A' L' R' W'#is_{j}'' and
is_{j}''': instrs zs @ is_{sbj} = is_{j}''' @ prog-instrs zs

by clarsimp

from i-bound have i-bound-ys: i < length ?ts-ys

by auto

from i-bound' neq-i-j

have ts-ys-i: ?ts-ys!i = (p_{sb}, is_{sbj}',

\theta_{sb}(t \mapsto \text{ret} (m a) (f (\theta_{sb}(t \mapsto m a)))),(), False, C_{sb} \cup A - R,\text{Map.empty})

by simp

from j-bound-ts-a have j-bound-ys: j < length ?ts-ys

by auto

then have ts-ys-j: ?ts-ys!j = (last-prog (hd-prog p j zs) ys, Write True a' sop' A' L' R'
W'#is_{j}'', \theta_{sbj} | (dom \theta_{sbj} - read-tmps zs), (), D_{j} \cup \text{outstanding-refs is-volatile-Write}_{sb} ys

\neq \{\},

acquired True ys (acquired True ?take-sb_{j} C_{j},R_{j}')

by (clarsimp simp add: is_{j}')

note conflict-computation = r-tranclp-rtranclp [OF step-a steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]

559
have safe-delayed (?ts-ys, ?m-ys, ?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j]
have a-unowned:
\( \forall i < \text{length } ts. \; j \neq i \rightarrow (let (O_i) = \text{map owned } ?ts-ys!i \text{ in } a' \notin O_i) \)
apply cases
apply (auto simp add: Let-def)
done
from a-in a-unowned [rule-format, of i] neq-i-j i-bound show False
by (auto simp add: Let-def)
qed
qed

thus ?thesis
by (auto simp add: Let-def)
qed

have A-unacquired-by-others:
\( \forall j < \text{length } (\text{map } O\text{-sb } ts_{sb}). \; i \neq j \rightarrow \)
\( (let (O_j, sb_j) = \text{map } O\text{-sb } ts_{sb}!j \text{ in } A \cap \text{all-acquired } sb_j = \{\}) \)
proof -
{
  fix j O_j sb_j
  assume j-bound: j < \text{length } (\text{map owned } ts_{sb})
  assume neq-i-j: i \neq j
  assume ts_{sb}-j: (map O\text{-sb } ts_{sb})!j = (O_j, sb_j)
  assume conflict: A \cap \text{all-acquired } sb_j \neq \{\}
  have False
  proof -
  from j-bound leq
  have j-bound': j < \text{length } (\text{map owned } ts)
  by auto
  from j-bound have j-bound'': j < \text{length } ts_{sb}
  by auto
  from j-bound' have j-bound'''': j < \text{length } ts
  by simp
  from conflict obtain a' where
  a'-in: a' \in A and
  a'-in-j: a' \in \text{all-acquired } sb_j
  by auto
  let ?take-sb_j = (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j)
  let ?drop-sb_j = (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j)
  from ts-sim [rule-format, OF j-bound'''] ts_{sb} \cdot j \cdot \text{bound}'''
  obtain p_j suspends_j is_{sb_j} \partial_{sb_j} D_{sb_j} R_j D_j is_j where
ts_{sb-j}: ts_{sb} \uparrow j = (p_j, is_{sbj}, \theta_{sbj}, sb_j, D_{sbj}, O_j, R_j) \text{ and suspend}_{sbj} = ?\text{drop-sb}_{sbj} \text{ and }

is_j: instrs suspend_{sbj} @ is_{sbj} = is_j @ prog-instrs suspend_{sbj} \text{ and }

D_j: D_{sbj} = (D_j \lor \text{outstanding-refs is-volatile-Write}_{sbj} \neq \{\}) \text{ and }

ts_j: ts_{sbj} = (hd-prog p_j suspend_{sbj}, is_j, 

\theta_{sbj} | (\text{dom } \theta_{sbj} \rightarrow \text{read-tmps suspend}_{sbj}), () , 

D_j, acquired True ?take-sb_{sbj} O_j, release ?take-sb_{sbj} (\text{dom } S_{sbj} R_j)

apply \text{ (cases ts}_{sb_{sb-j}})

apply (\text{force simp add: Let-def})

done

from a’-in-j all-acquired-append [of ?take-sb_{sbj} ?drop-sb_{sbj}]

have a’ \in all-acquired ?take-sb_{sbj} \lor a’ \in all-acquired suspend_{sbj}

by (auto simp add: suspend_{sbj})

thus False

proof

assume a’ \in all-acquired ?take-sb_{sbj}

with A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb-j} j-bound a’-in

show False

by (auto dest: all-acquired-unshared-acquired)

next

assume conflict-drop: a’ \in all-acquired suspend_{sbj}

from split-all-acquired-in [OF conflict-drop]

show ?thesis

proof

assume \exists sop a’’ v y z A’ L’ R’ W’.

suspend_{sbj} = y s @ Write_{sbj} True a’’ sop v A’ L’ R’ W’ # z s \land a’ \in A

then

obtain a’’ sop’ v’ y z A’ L’ R’ W’ where

split-suspend_{sbj}: suspend_{sbj} = y s @ Write_{sbj} True a’’ sop’ v’ A’ L’ R’ W’ # z s (is suspend_{sbj}) = ?suspend_{sbj}) \text{ and }

a’-A’, a’ \in A

by blast

from valid-program-history [OF j-bound”’ ts_{sb-j}]

have causal-program-history is_{sbj} sb_j.

then have cph: causal-program-history is_{sbj} suspend_{sbj}

apply –

apply (rule causal-program-history-suffix [where sb=?take-sb_{sbj} ] )

apply (simp only: split-suspend_{sbj} [symmetric] suspend_{sbj})

apply (simp add: split-suspend_{sbj})

done

from valid-last-prog [OF j-bound”’ ts_{sb-j} ] have last-prog: last-prog p_j sb_j = p_j, then

have lp: last-prog p_j suspend_{sb} = p_j

apply –
apply (rule last-prog-same-append [where sb=?take-sb])
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply simp
done

from valid-reads [OF j-bound'' ts_{sb}-j]
have reads-consis: reads-consistent False O_j m_{sb} sb_j.
from reads-consistent-flush-all-until-volatile-write [OF 
(valid-ownership-and-sharing S_{sb} ts_{sb}) j-bound''
ts_{sb}-j this]
have reads-consis-m-j: 
  reads-consistent True (acquired True ?take-sb_j O_j) m suspends_j 
  by (simp add: m suspends_j)

from outstanding-non-volatile-ref-owned-or-read-only [OF j-bound'' ts_{sb}-j]
have nvo-j: non-volatile-owned-or-read-only False S_{sb} O_j sb_j.
with non-volatile-owned-or-read-only-append [of False S_{sb} O_j ?take-sb_j ?drop-sb_j]
have nvo-take-j: non-volatile-owned-or-read-only False S_{sb} O_j ?take-sb_j 
  by auto

from a-unowned-others [rule-format, OF - neq-i-j] ts_{sb}-j j-bound
have a-not-acq: a /∈ acquired True ?take-sb_j O_j 
  by auto

from a-notin-unforwarded-non-volatile-reads-drop[OF j-bound'' ts_{sb}-j neq-i-j]
have a-notin-unforwarded-reads: a /∈ unforwarded-non-volatile-reads suspends_j {} 
  by (simp add: suspends_j)

let ?ma= (m(a := f (\theta_{sb}(t\mapsto m a))))

from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads [where W={}
  and m'=?ma,simplified, OF - subset-refl reads-consis-m-j]
a-notin-unforwarded-reads
have reads-consis-ma-j: 
  reads-consistent True (acquired True ?take-sb_j O_j) ?ma suspends_j 
  by auto

from reads-consis-ma-j
have reads-consis-ys: reads-consistent True (acquired True ?take-sb_j O_j) ?ma (ys) 
  by (simp add: split-suspends_j reads-consistent-append)

from direct-memop-step.RMWWrite [where cond=cond and \varnothing=\emptyset_{sb} and m=m, OF cond']
have (RMW a t (D, f) cond ret A L R W# i_{sb}', 
  i_{sb} (\lambda), m, D, O_{sb}, R_{sb}, S) \rightarrow 
  (i_{sb}', 
  i_{sb}(t \mapsto ret (m a) (f (i_{sb}(t \mapsto m a))))), (\lambda), ?ma, False, O_{sb} \cup A - R, Map.empty, S 
  \oplus_W R \ominus_A L).

562
from direct-computation.concurrent-step.

have step-a: (ts, m, S) ⇒ₚ
  (ts[i := (pₛ_b, iₛ_b', θₛ_b(t → ret (m a) (f (θₛ_b(t → m a))))), ()), False, Cₛ_b ∪ A − R, Map.empty)
  (is - ⇒ₜ ?ts-a, - (sэт-shared-a)).

from tsⱼ neq-i-j

have ts-a-j: ?ts-a!ⱼ = (hd-prog pⱼ suspendsⱼ, iⱼ, θⱼ, Dⱼ, acquired True ?take-sbⱼ @?suspendsⱼ, release ?take-sbⱼ (dom Sₛ_b) Rⱼ)
  by auto

from valid-write-sops [OF j-bound'' tsₛ_b-j]

have ∀ sop∈write-sops (?take-sbⱼ@?suspendsⱼ). valid-sop sop
  by (simp add: split-suspendsⱼ [symmetric] suspendsⱼ)
then obtain valid-sops-take: ∀ sop∈write-sops ?take-sbⱼ. valid-sop sop and
  valid-sops-drop: ∀ sop∈write-sops (ys). valid-sop sop
  apply (simp only: write-sops-append)
  apply auto
  done

from read-tmps-distinct [OF j-bound'' tsₛ_b-j]

have distinct-read-tmps (?take-sbⱼ@?suspendsⱼ)
  by (simp add: split-suspendsⱼ [symmetric] suspendsⱼ)
then obtain
  read-tmps-take-drop: read-tmps ?take-sbⱼ ∩ read-tmps suspendsⱼ = {}
  and
  distinct-read-tmps-drop: distinct-read-tmps suspendsⱼ
  apply (simp only: split-suspendsⱼ [symmetric] suspendsⱼ)
  apply (simp only: distinct-read-tmps-append)
  done

from valid-history [OF j-bound'' tsₛ_b-j]

have h-consis:
  history-consistent θⱼ (hd-prog pⱼ (?take-sbⱼ@?suspendsⱼ)) (?take-sbⱼ@?suspendsⱼ)
  apply (simp only: split-suspendsⱼ [symmetric] suspendsⱼ)
  apply simp
  done

have last-prog-hd-prog: last-prog (hd-prog pⱼ sbⱼ) ?take-sbⱼ = (hd-prog pⱼ suspendsⱼ)
proof
  from last-prog have last-prog pⱼ (?take-sbⱼ@?drop-sbⱼ) = pⱼ
    by simp
  from last-prog-hd-prog-append' [OF h-consis] this
  have last-prog (hd-prog pⱼ suspendsⱼ) ?take-sbⱼ = hd-prog pⱼ suspendsⱼ
    by (simp only: split-suspendsⱼ [symmetric] suspendsⱼ)

563
moreover
have last-prog (hd-prog p j (?take-sb j @ suspends j)) ?take-sb j =
last-prog (hd-prog p j suspends j) ?take-sb j
apply (simp only: split-suspends j [symmetric] suspends j)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends j [symmetric] suspends j)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
h-consis] last-prog-hd-prog
have hist-consis': history-consistent \( \delta_{\text{sbj}} \) (hd-prog p j suspends j) suspends j
by (simp add: split-suspends j [symmetric] suspends j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read_{ab} (ys) = {}
by (auto simp add: outstanding-refs-append suspends j [symmetric]
split-suspends j)
from j-bound' have j-bound-ts-a: j < length ?ts-a by auto

from flush-store-buffer-append [where sb=ys and sb'=?Write_{ab} True a'' sop' v' A' L' R'
W'#@zs, simplified,]
of j-bound-ts-a \( \text{is}_j \) [simplified split-suspends j] cph [simplified suspends j]
hist-consis' [simplified split-suspends j] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends j]
no-volatile-Read_{ab}-volatile-reads-consistent [OF no-vol-read], where
\( S=\text{?shared-a} \)

obtain \( \text{is}_j', \mathcal{R}_j' \) where
\( \text{is}_j' \): Write True a'' sop' A' L' R' W'\#@zs, \( \text{is}_j' @ prog-instrs zs \) and
steps-ys: (?ts-a, ?ma, ?shared-a) \( \Rightarrow_d^* \)
(?ts-aj:=(last-prog
  (hd-prog p j zs) ys,
  \( \delta_{\text{sbj}} \),
  \( \text{is}_j' \),
  \( \delta_{\text{sbj}} \),
  \( \text{dom} \ \delta_{\text{sbj}} - \text{read-tmps} @zs \),
  \),
\( D_j \lor \text{outstanding-refs is-volatile-Write}_{ab} @zs \neq \{\}, \) acquired True @zs (acquired
True ?take-sb j \( O_j, \mathcal{R}_j' \)),
flush ys (?ma),
share ys (?shared-a))
(is (\_,\_,\_) \( \Rightarrow_d^* \) (?ts-ys,?m-ys,?shared-ys))
by (auto simp add: acquired-append)

from cph
have causal-program-history \( \text{is}_{\text{sbj}} \) ((ys @ [Write_{ab} True a'' sop' v' A' L' R' W']) @ zs)
by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history \( \text{is}_{\text{sbj}} @zs \).
interpret causal:j: causal-program-history \( \text{is}_{\text{sbj}} @zs \) by (rule cph')

564
from causal-program-history [of [], simplified, OF refl] is_j'

obtain is_j''
  where is_j': is_j' = Write True a'' sop' A' L' R' W'#is_j'' and
  is_j'': instrs zs @ is_{sbj} = is_j'' @ prog-instrs zs
  by clarsimp

from i-bound' have i-bound-ys: i < length ?ts-ys
  by auto

from i-bound' neq-i-j
have ts-ys-i: ?ts-ys!i = (p_{sb}, is_{sb}'.
  \vartheta_{sb}(t \mapsto \text{ret (m a)}) (f (\vartheta_{sb}(t \mapsto m a))), (), False, O_{sb} \cup A - R, Map.empty)
  by simp

from j-bound-ts-a have j-bound-ys: j < length ?ts-ys
  by auto
then have ts-ys-j: ?ts-ys!j = (last-prog (hd-prog p_j zs) ys, Write True a'' sop' A' L' R'
  W'#is_j''
  \vartheta_{sbj} | (\vartheta_{sbj} - \text{read-tmps zs}), ()
  D_j \lor \text{outstanding-refs is-volatile-Write}_{sb} ys \neq \{\}
  acquired True ys (acquired True ?take-sb \vartheta_j, R_j)
  by (clarsimp simp add: is_{sb}'')
  note conflict-computation = r-rtranclp-rtranclp [OF step-a steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j]
have A'-unowned:
  \forall i < length ?ts-ys. j\neq i \longrightarrow (let (O_i) = map owned ?ts-ys!i in A' \cap O_i = \{\})
  apply cases
  apply (fastforce simp add: Let-def is_{sb})+
done
from a'-in a'-A' A'-unowned [rule-format, of i] neq-i-j i-bound' A-R
show False
  by (auto simp add: Let-def)
next
assume \exists A L R W ys zs. suspends_j = ys @ Ghost_{sb} A L R W# zs \land a' \in A
then obtain ys zs A' L' R' W' where
  split-suspends_j: suspends_j = ys @ Ghost_{sb} A' L' R' W'# zs (is suspends_j = ?suspends)
and
  a'-A'; a' \in A'
  by blast

from valid-program-history [OF j-bound'' ts_{sb}-j]
have causal-program-history is_{sbj} sb_j.
then have cph: causal-program-history is \( \text{sbj} \) suspends
  apply –
  apply (rule causal-program-history-suffix [\text{where} \ sb = ?\text{take-sbj}] )
  apply (simp only:\ split-suspends_j [symmetric] suspends_j)
  apply (simp add:\ split-suspends_j)
  done

from valid-last-prog [OF \ j-bound'' ts_{sb-j}] have last-prog: last-prog \( p_j \) \( \text{sbj} \) = \( p_j \).
then
  have lp: last-prog \( p_j \) ?suspends = \( p_j \)
  apply –
  apply (rule last-prog-same-append [\text{where} \ sb = ?\text{take-sbj}] )
  apply (simp only:\ split-suspends_j [symmetric] suspends_j)
  apply simp
  done

from valid-reads [OF \ j-bound'' ts_{sb-j}]
  have reads-consis: reads-consistent False \( O_j \) \( m_{sb} \) \( \text{sbj} \).
from reads-consistent-flush-all-until-volatile-write [OF
  \langle valid-ownership-and-sharing \( S_{sb} \) ts_{sb} \rangle \ j-bound''
  \( ts_{sb-j} \) this]
  have reads-consis-m-j:
    reads-consistent True (acquired True ?\text{take-sbj} \( O_j \)) \( m \) suspends_j
    by (simp add: \( m \) suspends_j)
from outstanding-non-volatile-refs-owned-or-read-only [OF \ j-bound'' ts_{sb-j}]
  have nvo-j: non-volatile-owned-or-read-only False \( S_{sb} \) \( O_j \) \( \text{sbj} \).
  with non-volatile-owned-or-read-only-append [of False \( S_{sb} \) \( O_j \) ?\text{take-sbj} ?\text{drop-sbj}]
  have nvo-take-j: non-volatile-owned-or-read-only False \( S_{sb} \) \( O_j \) ?\text{take-sbj}
    by auto
from a-unowned-others [rule-format, OF - neq-i-j] ts_{sb-j} \ j-bound
  have a-not-acq: \( a \notin \) acquired True ?\text{take-sbj} \( O_j \)
    by auto
from a-notin-unforwarded-non-volatile-reads-drop[OF \ j-bound'' ts_{sb-j} neq-i-j]
  have a-notin-unforwarded-reads: \( a \notin \) unforwarded-non-volatile-reads suspends_j \{\}
    by (simp add: suspends_j)
let \( ?m_a = (m(a := f (\theta_{sb}(b \mapsto m a))))\)
from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads [\text{where} \ W = \{\} and \( m' = ?m_a \), simplified, OF - subset-refl reads-consis-m-j]
  a-notin-unforwarded-reads
have reads-consis-ma-j:
  reads-consistent True (acquired True ?\text{take-sbj} \( O_j \)) \( ?m_a \) suspends_j
  by auto

566
from reads-consis-ma-j
have reads-consis-ys: reads-consistent True (acquired True ?take-sb_j O_j) ?ma (ys)
by (simp add: split-suspends_j reads-consistent-append)

from direct-menop-step.RMWWrite [where cond=cond and \vartheta=\vartheta_{sb} and m=m, OF cond]
have (RMW a t (D, f) cond ret A L R W\# i_{sb}',
\vartheta_{sb}(\cdot, m, D, O_{sb}, R_{sb}, S) \rightarrow
\vartheta'(t \mapsto \text{ret} (m a) (f (\vartheta_{sb}(t \mapsto m a)))), (), \text{?ma}, \text{False}, O_{sb} \cup A - R, Map.empty; S \oplus W R \ominus A L).
from direct-computation.concurrent-step.Memop [OF i-bound ts-i [simplified sb, simplified] this]
have step-a: (ts, m, S) \Rightarrow_d (ts[i := (p sb, i_{sb}', \vartheta_{sb}(t \mapsto \text{ret} (m a) (f (\vartheta_{sb}(t \mapsto m a)))), ()], \text{?ma}, \text{False}, O_{sb} \cup A - R, Map.empty),
\vartheta_{sb} (\cdot) (\text{is} \{-\Rightarrow_d (?ts-a, -, ?shared-a)}).

from ts_j neq-i-j j-bound
have ts-a-j: ?ts-alj = (hd-prog p_j suspends_j, is_j,
\vartheta_{sbj} \mid (\text{dom} \vartheta_{sbj} - \text{read-tmps suspends}_j)(\cdot), D_j, \text{acquired True ?take-sb_j O_j, release}
?take-sb_j (\text{dom} S_{sbj} R_j))
by auto

from valid-write-sops [OF j-bound'' ts_{sb-j}]
have \forall \text{sop} \in \text{write-sops} (?take-sb_j @ ?\text{suspends}_j). \text{valid-sop sop}
by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain valid-sops-take: \forall \text{sop} \in \text{write-sops} ?take-sb_j. \text{valid-sop sop and}
valid-sops-drop: \forall \text{sop} \in \text{write-sops} (ys). \text{valid-sop sop}
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound'' ts_{sb-j}]
have distinct-read-tmps (?take-sb_j @ ?suspends_j)
by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain read-tmps-take-drop: read-tmps ?take-sb_j \cap \text{read-tmps suspends}_j = \{\} and
distinct-read-tmps-drop: distinct-read-tmps suspends_j
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound'' ts_{sb-j}]
have h-consis:
history-consistent \vartheta_{sbj} (hd-prog p_j (?take-sb_j @ suspends_j)) (?take-sb_j @ suspends_j)
apply (simp only: split-suspends \[symmetric\] suspends\[symmetric\])
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p\[j\] sb\[j\]) ?take-sb\[j\] = (hd-prog p\[j\] suspends\[j\])
proof
  from last-prog have last-prog p\[j\] (?take-sb\[j\]@drop-sb\[j\]) = p\[j\]
  by simp
from last-prog-hd-prog-append’ [OF h-consis] this
have last-prog (hd-prog p\[j\] suspends\[j\]) ?take-sb\[j\] = hd-prog p\[j\] suspends\[j\]
  by (simp only: split-suspends \[symmetric\] suspends\[symmetric\])
moreover
have last-prog (hd-prog p\[j\] suspends\[j\]) ?take-sb\[j\] =
  last-prog (hd-prog p\[j\] suspends\[j\]) ?take-sb\[j\]
apply (simp only: split-suspends \[symmetric\] suspends\[symmetric\])
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends \[symmetric\] suspends\[symmetric\])
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog
have hist-consis': history-consistent \[\theta\]sbj (hd-prog p\[j\] suspends\[j\]) suspends\[j\]
  by (simp add: split-suspends \[symmetric\] suspends\[symmetric\])
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read\sb (ys) = {}
  by (auto simp add: outstanding-refs-append suspends \[symmetric\] split-suspends\[symmetric\])
from j-bound' have j-bound-ts-a: j < length \[?ts-a\] by auto

from flush-store-buffer-append [where \[sb=ys\] and \[sb'=Ghost\sb\ A' L' R' W'\#zs, simplified, OF j-bound-ts-a is\[j\] [simplified split-suspends\[j\]] cph [simplified suspends\[j\]] ts-a-j [simplified split-suspends\[j\]] refl lp [simplified split-suspends\[j\]] reads-consis-ys
  hist-consis' [simplified split-suspends\[j\]] valid-sops-drop
  distinct-read-tmps-drop [simplified split-suspends\[j\]] no-volatile-Read\sb-volatile-reads-consistent [OF no-vol-read], where \[S=?shared-a\]

obtain is\[j\]' \[R\]j' where
  is\[j\]' : Ghost A' L' R' W'\# instrs zs @ is\sbj = is\[j\]' @ prog-instrs zs and
  steps-ys: (?ts-a, ?ma, ?shared-a) \Rightarrow_d^*?
  (?ts-a[j]:=\{last-prog
    (hd-prog p\[j\] zs) ys,
    is\[j\]',
    \[\theta\]sbj \mid (dom \[\theta\]sbj - read-tmps zs),
    ()
  \},
  D\[j\] \lor outstanding-refs is-volatile-Write\sb (ys) \neq \{\}, acquired True ys (acquired
  True ?take-sb\[j\] O\[j\],R\[j\]'))]},

568
flush ys (?ma),
share ys (?shared-a))
(is (¬¬¬) \Rightarrow d∗ (?ts-ys,?m-ys,?shared-ys))
by (auto simp add: acquired-append)

from cph
have causal-program-history is sbj (ys @ [Ghost sb A′ L′ R′ W′] @ zs)
by simp
from causal-program-history-suffix [OF this]
have cph′: causal-program-history is sbj zs.
interpret causal1: causal-program-history is sbj zs by (rule cph′)

from causal1,causal-program-history [of [], simplified, OF refl] isj′
obtain isj′′
where isj′′ = Ghost A′ L′ R′ W′#isj′′ and
isj′′: instrs zs @ is sbj = isj′′ @ prog-instrs zs
byclarsimp

from i-bound′ have i-bound-ys: i < length ?ts-ys
by auto

from i-bound′ neq-i-j
have ts-ys-i: ?ts-ys!i = (p sb, is sb′, θ sb (t \mapsto \rightarrow ret (m a) (f (θ sb (t \mapsto \rightarrow m a)))), (), False, O sb ∪ A − R,Map.empty)
by simp

from j-bound-ts-a have j-bound-ys: j < length ?ts-ys
by auto
then have ts-ys-j: ?ts-ys!j = (last-prog (hd-prog p j zs) ys, Ghost A′ L′ R′ W′#isj′′, 
\varnothing (t \mapsto \rightarrow ret (m a)) (f (\varnothing (t \mapsto \rightarrow m a))), ()), False, O sb ∪ A − R,Map.empty)
by simp

from safe-delayedE [OF this j-bound-ys ts-ys-j]
have \exist i < length ?ts-ys. j \neq i \rightarrow (let (O i) = map owned ?ts-ys!i in A′ ∩ O i = {})
apply cases
apply (fastforce simp add: Let-def is sb)+
done
from a′-in a′A′ A′-unowned [rule-format, of i neq-i-j i-bound′ A-R
show False
by (auto simp add: Let-def)
qed
qed
thus \textit{thesis}

by (auto simp add: Let-def)

qed
have causal-program-history is_{sbj} sb_j.

then have cph: causal-program-history is_{sbj} ?suspends

apply –
apply (rule causal-program-history-suffix [where sb=?take-sb_j] )
apply (simp only: split-suspends_{sbj} [symmetric] suspends_{sbj})
apply (simp add: split-suspends_{sbj})
done

from valid-last-prog [OF j-bound ts_{sbj}] have last-prog: last-prog p_j sb_j = p_j.

then
have lp: last-prog p_j ?suspends = p_j
apply –
apply (rule last-prog-same-append [where sb=?take-sb_j])
apply (simp only: split-suspends_{sbj} [symmetric] suspends_{sbj})
apply simp
done

from valid-reads [OF j-bound ts_{sbj}] have reads-consis: reads-consistent False Ω_j m_{sb} sb_j.

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing Ω_{sb} ts_{sb} j-bound ts_{sbj} this]

have reads-consis-m-j: reads-consistent True (acquired True ?take-sb_j Ω_j) m suspends_{sbj}
by (simp add: m suspends_{sbj})

from outstanding-non-volatile-refs-owned-or-read-only [OF j-bound ts_{sbj}]

have nvo-j: non-volatile-owned-or-read-only False Ω_{sb} Ω_j sb_j.

with non-volatile-owned-or-read-only-append [of False Ω_{sb} Ω_j ?take-sb_j ?drop-sb_j]

have nvo-take-j: non-volatile-owned-or-read-only False Ω_{sb} Ω_j ?take-sb_j
by auto

from a-unowned-others [rule-format, OF - neq-i-j] ts_{sbj} j-bound

have a-not-acq: a /∈ acquired True ?take-sb_j Ω_j
by auto

from a-notin-unforwarded-non-volatile-reads-drop[OF j-bound ts_{sbj} j neq-i-j]

have a-notin-unforwarded-reads: a /∈ unforwarded-non-volatile-reads suspends_{sbj} {}
by (simp add: suspends_{sbj})

let ?ma=(m(a := f (θ_{sb}(t→m a))))

from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads [where W={}]
and m'=?ma,simplified, OF - subset-refl reads-consis-m-j]
a-notin-unforwarded-reads

have reads-consis-ma-j:
reads-consistent True (acquired True ?take-sb_j Ω_j) ?ma suspends_{sbj}
by auto

from reads-consis-ma-j
have reads-consis-ys: reads-consistent True (acquired True ?take-sb \(j\) ?ma (ys)
by (simp add: split-suspends\(j\) reads-consistent-append)

from direct-memop-step.RMWWrite [where cond=cond and \(\emptyset=\emptyset_{sb}\) and \(m=m\), OF cond']
have (RMW a (D, f) cond ret A L R \# \(\emptyset_{sb}'\), \(\emptyset_{sb}\), (), m, \(D, O_{sb}, R_{sb}, S\)) \(\Rightarrow\)
\((is_{sb}', \emptyset_{sb}(t \mapsto ret (m a) (f (\emptyset_{sb}(t \mapsto m a)))), ()\), ?ma, False, \(O_{sb} \cup A - R, Map.empty, S \oplus W R \ominus A L)\).
from direct-computation.concurrent-step.Memop [OF i-bound' ts-i this]
have step-a: \((ts, m, S) \Rightarrow\)
\((ts[i := (p_{sb}, is_{sb}'}, \emptyset_{sb}(t \mapsto ret (m a) (f (\emptyset_{sb}(t \mapsto m a)))), ()\), False, \(O_{sb} \cup A - R, Map.empty)]\),
\(?ma, S \oplus W R \ominus A L)\)
(is - \(\Rightarrow\) \((?ts-a, -, \?shared-a)\)).

from ts\(_j\) neq-i-j j-bound

have ts-a-j: \(?ts-a!j = (hd-prog p\(_j\) suspends\(j\), is\(_j\),
\(\emptyset_{sbj} \mid (dom \emptyset_{sbj} - read-tmps suspends\(j\)),(), D\(_j\), acquired True ?take-sb \(O_j, release ?take-sb \(j\) (dom S_{sb}) R_j)\)
by auto

from valid-write-sops [OF j-bound ts\(_{sb-j}\)]
have \(?\forall s\in write-sops (?take-sb \(j\) ?\(\emptyset_{suspends}\)). valid-sop s\)
by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\) )
then obtain valid-sops-take: \(?\forall s\in write-sops ?take-sb \. valid-sop s\)
valid-sops-drop: \(?\forall s\in write-sops (ys). valid-sop s\)
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound ts\(_{sb-j}\)]
have \(?\forall s\in read-tmps (?take-sb \(j\) ?\(\emptyset_{suspends}\)). distinct-read-tmps\)
by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\) )
then obtain
read-tmps-take-drop: \(read-tmps ?take-sb \cap read-tmps suspends\(_j\) = {}\)
and
distinct-read-tmps-drop: \(distinct-read-tmps suspends\(_j\)\)
apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\) )
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound ts\(_{sb-j}\)]
have \(?history-consistent \emptyset_{sbj} (hd-prog p\(_j\) (?take-sb \(j\) ?\(\emptyset_{suspends}\))). \(\emptyset_{take-sb \(j\) \(\emptyset_{suspends}\)\)
apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\) )
apply simp
done

have last-prog-hd-prog: \(last-prog (hd-prog p\(_j\) sb\(_j\) ?take-sb \(_j\) = (hd-prog p\(_j\) suspends\(_j\) )\)
proof
from last-prog have last-prog \( p_j \) (?\( \text{take-sb} @ \text{drop-sb} \)) = \( p_j \)
by simp
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog \( p_j \) suspends\( j \)) ?\( \text{take-sb} \) = hd-prog \( p_j \) suspends\( j \)
by (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
moreover
have last-prog (hd-prog \( p_j \) (?\( \text{take-sb} \) @ suspends\( j \))) ?\( \text{take-sb} \) = 
last-prog (hd-prog \( p_j \) suspends\( j \)) ?\( \text{take-sb} \)
apply (simp only: split-suspends\( j \) [symmetric] suspends\( j \))
by (rule last-prog-hd-prog-append)
ultimately show \(?\text{thesis}\)
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog
have hist-consis': history-consistent \( \vartheta_{sbj} \) (hd-prog \( p_j \) suspends\( j \)) suspends\( j \)
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
from reads-consistent-drop-volatile-writes-no-volatile-reads [OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read\( sb \) (ys) = {}
by (auto simp add: outstanding-refs-append suspends\( j \) [symmetric] split-suspends\( j \))
from j-bound leq have j-bound-ts-a: \( j \) < length ?\( ts-a \) by auto

from flush-store-buffer-append [where sb=ys and sb'='Read\( sb \) False \( t' \) \( v' \)\#zs, simplified,
OF \( j \)-bound-ts-a is\( j \) [simplified split-suspends\( j \)] cph [simplified suspends\( j \)]
\( ts-a-j \) [simplified split-suspends\( j \)] refl lp [simplified split-suspends\( j \)] reads-consis-ys
hist-consis' [simplified split-suspends\( j \)] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends\( j \)]
no-volatile-Read\( sb \)-volatile-reads-consistent [OF no-vol-read], where
\( S=\?\text{shared-a} \)

obtain is\( j \) ' \( R_j ' \) where
is\( j \)' : Read False a' \( t' \)'# instrs zs @ is\( sbj \) = is\( j \)' @ prog-instrs zs and
steps-ys: (?\( ts-a \), \( ?\text{ma}, \?\text{shared-a} \) \( \Rightarrow d^* \)
(?\( ts-a-j \)=: (last-prog \( (hd-prog \( p_j \) zs) \) ys,
, is\( j \)',
, \( \vartheta_{sbj} ' \) (dom \( \vartheta_{sbj} ' \) - insert \( t' \) (read-tmps zs)),
, (), \( D_j \) \lor outstanding-refs is-volatile-Write\( sb \) ys \( \neq \) {}, acquired True ys (acquired
True ?\( \text{take-sb} \) Q\( j \),\( R_j ' \)],
flush ys (?\( \text{ma} \),
share ys (?\( \text{shared-a} \))
(is (-,-,-) \( \Rightarrow d^* \) (?\( ts-ys \),?\( m-ys,?\text{shared-ys} \))

573
by (auto simp add: acquired-append)

from cph
have causal-program-history is_{sbj} ((ys @ [Read_{sb} False a' t' v']) @ zs)
by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history is_{sbj} zs.
interpret causal_{j}: causal-program-history is_{sbj} zs by (rule cph')

from causal_{j}.causal-program-history [of [], simplified, OF refl] is_{j}'
obtain is_{j}''
where is_{j}': is_{j}' = Read False a' t'#is_{j}'' and
is_{j}'' : instrs zs @ is_{sbj} = is_{j}'' @ prog-instrs zs
by clarsimp

from i-bound' have i-bound-ys: i < length ?ts-ys
by auto

from i-bound' neq-i-j
have ts-ys-i: ?ts-ys!i = (p_{sb}, is_{sb}'',
θ sb(t↦→ ret (m a) (f (θ sb(t ↦→ m a)) ),(), False, O_{sb} ∪ A − R, Map.empty)
by simp

from j-bound-ts-a have j-bound-ys: j < length ?ts-ys
by auto
then have ts-ys-j: ?ts-ys!j = (last-prog (hd-prog p_{j} zs) ys, Read False a' t'#is_{j}''', θ sbj
|′ (dom θ sbj − insert t' (read-tmps zs)), ( ), D_{j} ∨ outstanding-refs is-volatile-Write_{sb} ys ≠
{}, acquired True ys (acquired True ?take-sb_{j} O_{j}), R_{j}')
by (clarsimp simp add: is_{sb})
note conflict-computation = r-rtranclp-rtranclp [OF step-a steps-ys]

from safe-reach-safe-rtranclp [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j]

have a' ∈ acquired True ys (acquired True ?take-sb_{j} O_{j}) ∨
a' ∈ read-only (share ys (S ⊕ W R ⊕ A L))
apply cases
apply (auto simp add: Let-def is_{sb})
done

with a'-'unacq
have a'-'ro: a' ∈ read-only (share ys (S ⊕ W R ⊕ A L))
by auto
from a'-'in
have a'-'not-ro: a' /∈ read-only (S ⊕ W R ⊕ A L)
by (auto simp add: in-read-only-convs)
have \( a' \in O_j \cup \text{all-acquired } sb_j \)

proof -

{ 
  \begin{aligned}
    \text{assume} & \quad \text{a-notin: } a' \notin O_j \cup \text{all-acquired } sb_j \\
    \text{from} & \quad \text{weak-sharing-consis } \{ \text{OF } j\text{-bound } ts_{sb-j} \} \\
    \text{have} & \quad \text{weak-sharing-consistent } O_j sb_j, \\
    \text{with} & \quad \text{weak-sharing-consistent-append } \{ \text{of } O_j ?\text{take-sb}_{j} ?\text{drop-sb}_{j} \} \\
    \text{have} & \quad \text{weak-sharing-consistent } (\text{acquired True } ?\text{take-sb}_{j} O_j) \text{ suspends}_{j} \\
    \quad & \quad \text{by } (\text{auto simp add: suspends}_{j}) \\
    \text{with} & \quad \text{split-suspends}_{j} \\
    \text{have} & \quad \text{weak-consis: weak-sharing-consistent } (\text{acquired True } ?\text{take-sb}_{j} O_j) \text{ ys} \\
    \quad & \quad \text{by } (\text{simp add: weak-sharing-consistent-append}) \\
    \text{from} & \quad \text{all-acquired-append } \{ ?\text{take-sb}_{j} ?\text{drop-sb}_{j} \} \\
    \text{have} & \quad \text{all-acquired } \text{ys } \subseteq \text{all-acquired } sb_j \\
    \quad & \quad \text{apply } (\text{clarsimp}) \\
    \quad & \quad \text{apply } (\text{clarsimp simp add: suspends}_{j} \text{ symmetric} \text{ split-suspends}_{j} \text{ all-acquired-append}) \\
    \quad \text{done} \\
    \quad \text{with} & \quad \text{a-notin acquired-takeWhile-non-volatile-Write}_{sb} \quad \{ \text{of } sb_j O_j \} \\
    \quad \quad \text{all-acquired-append } \{ ?\text{take-sb}_{j} ?\text{drop-sb}_{j} \} \\
    \text{have} & \quad a' \notin \text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_{j}) \text{ } O_j \cup \text{all-acquired } \text{ys} \\
    \quad & \quad \text{by } \text{auto} \\
  \end{aligned}

\text{from } \text{read-only-share-unowned } \{ \text{OF weak-consis this } a'-\text{ro} \} \\
\text{have} \quad a' \in \text{read-only } (S \oplus W R \ominus A L) .

\text{with } a'-\text{not-ro have } \text{False} \\
\quad \text{by } \text{auto} \\
\text{with } a'-\text{notin read-only-share-unowned } \{ \text{OF weak-consis - } a'-\text{ro} \} \\
\quad \text{all-acquired-takeWhile } \{ \text{of } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_{j} \}

\text{have } a' \in \text{read-only } (S \oplus W R \ominus A L) \\
\quad \text{by } (\text{auto simp add: acquired-takeWhile-non-volatile-Write}_{sb}) \\
\text{with } a'-\text{not-ro have } \text{False} \\
\quad \text{by } \text{auto} \\
}

\text{thus } \text{?thesis by blast} \\
\text{qed}

\text{moreover} \\
\text{from } \text{A-unacquired-by-others } \{ \text{rule-format, OF - neq-i-j} \} ts_{sb-j} \text{ j-bound} \\
\text{have } A \cap \text{all-acquired } sb_j = \{ \}
\quad \text{by } (\text{auto simp add: Let-def})
\text{moreover} \\
\text{from } \text{A-unowned-by-others } \{ \text{rule-format, OF - neq-i-j} \} ts_{sb-j} \text{ j-bound} \\
\text{have } A \cap O_j = \{ \}
\quad \text{by } (\text{auto simp add: Let-def dest: all-shared-acquired-in})
\text{moreover note } a'-\text{in} \\
\text{ultimately}
show False
by auto
qed
}
thus ?thesis
by (auto simp add: Let-def)
qed

{ note A-no-read-only-reads = this

have valid-own': valid-ownership $S_{sb}'$ $ts_{sb}'$
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only $S_{sb}'$ $ts_{sb}'$
proof
fix $j$ is $j$ $O$ $j$ $R$ $j$ $D$ $j$ $θ$ $j$ $sb$ $j$ $p$ $j$
assume $j$-bound: $j < \text{length } ts_{sb}'$
assume $ts_{sb}'$-$j$: $ts_{sb}'$-$j$ = $(p_j, is_j, \vec{0}, sb_j, D_j, O_j, R_j)$
show non-volatile-owned-or-read-only False $S_{sb}'$ $O_j$ sb$_j$
proof (cases $j$=$i$)
case True
have non-volatile-owned-or-read-only False
$(S_{sb} \oplus W R \ominus A L) (O_{sb} \cup A - R) []$
by simp
then show ?thesis
using True $i$-bound $ts_{sb}'$-$j$
by (auto simp add: $ts_{sb}'$ $S_{sb}'$ $sb sb'$)
next
case False
from $j$-bound have $j$-bound': $j < \text{length } ts_{sb}'$
by (auto simp add: $ts_{sb}'$)
with $ts_{sb}'$-$j$ False $i$-bound
have $ts_{sb}$-$j$: $ts_{sb}$-$j$ = $(p_j, is_j, \vec{0}, sb_j, D_j, O_j, R_j)$
by (auto simp add: $ts_{sb}'$)

note nvo = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound’ $ts_{sb}$-$j$]

from read-only-unowned [OF i-bound $ts_{sb}$-$i$] R-owned
have $R \cap \text{read-only } S_{sb} = {}$
by auto
with A-no-read-only-reads [OF j-bound’ $ts_{sb}$-$j$ False [symmetric]] L-subset
have $\forall a \in \text{read-only-reads}$ [OF j-bound’ $ts_{sb}$-$j$ False [symmetric]] L-subset
  (acquired True (takeWhile (Not \circ is-volatile-Write$_{sb}$) sb$_j$) $O_j$)
  (dropWhile (Not \circ is-volatile-Write$_{sb}$) sb$_j$).
a $\in$ read-only $S_{sb} \rightarrow a \in$ read-only $(S_{sb} \oplus W R \ominus A L)$
by (auto simp add: in-read-only-convs)
from non-volatile-owned-or-read-only-read-only-reads-eq' [OF nvo this]
have non-volatile-owned-or-read-only False $(S_{sb} \oplus W R \ominus A L) O_j$ sb$_j$,
thus ?thesis by (simp add: $S_{sb}'$)
qed
qed

576
next
show outstanding-volatile-writes-unowned-by-others ts_{sb}′

proof (unfold-locales)
fix i_1 j p_1 is_1 O_1 R_1 D_1 xs_1 sb_1 p_j is_j O_j R_j D_j xs_j sb_j
assume i_1-bound: i_1 < length ts_{sb}′
assume j-bound: j < length ts_{sb}′
assume i_1-j: i_1 \neq j
assume ts-i: ts_{sb}′!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)
assume ts-j: ts_{sb}′!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
show (O_j \cup \text{all-acquired } sb_j) \cap \text{outstanding-refs is-volatile-Write}_{sb} sb_1 = {} 

proof (cases i_1=i)
case True
with ts-i_1 i-bound show ?thesis
by (simp add: ts_{sb}′ sb′ sb)
next
case False
note i_1-i = this
from i_1-bound have i_1-bound′: i_1 < length ts_{sb} by (simp add: ts_{sb}′ sb′ sb)
hence i_1-bound′′: i_1 < length (map owned ts_{sb})
by auto
from ts-i False have ts-i_1′: ts_{sb}′!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)
by (simp add: ts_{sb}′ sb′ sb)
show ?thesis
proof (cases j=i)
case True
from i-bound ts-j ts_{sb}′ True have sb_j: sb_j=[]
by (simp add: ts_{sb}′ sb′)
from A-unused-by-others [rule-format, OF - False [symmetric]] ts-i_1 i_1-bound′
False i_1-bound′
have A \cap (O_1 \cup \text{outstanding-refs is-volatile-Write}_{sb} sb_1) = {}
by (auto simp add: Let-def ts_{sb}′ sb′ sb′ owned-def)
moreover
from outstanding-volatile-writes-unowned-by-others
[OF i_1-bound′ i-bound i_1-i ts-i_1′ ts_{sb}′]
have O_{sb} \cap \text{outstanding-refs is-volatile-Write}_{sb} sb_1 = {} by (simp add: sb)
ultimately show ?thesis using ts-j True
by (auto simp add: i-bound ts_{sb}′ O_{sb}′ sb′)
next
case False
from j-bound have j-bound′: j < length ts_{sb}
by (simp add: ts_{sb}′)
from ts-j False have ts-j′: ts_{sb}′!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
by (simp add: ts_{sb}′)
from outstanding-volatile-writes-unowned-by-others
[OF i_1-bound′ j-bound′ i_1-j ts-i_1′ ts_{sb}′]
show (O_j \cup \text{all-acquired } sb_j) \cap \text{outstanding-refs is-volatile-Write}_{sb} sb_1 = {} .
qed
qed
next
show read-only-reads-unowned ts_{sb}\,'
proof
fix n m
fix p_n i_n \sigma_n \rho_n s_{b_n} p_m i_m \sigma_m \rho_m s_{b_m}
assume n-bound: n < length ts_{sb}\,'
and m-bound: m < length ts_{sb}\,'
and neq-n-m: n \neq m
and nth: ts_{sb}\,'
! n = (p_n, i_n, \rho_n, s_{b_n}, D_n, O_n, R_n)
and mth: ts_{sb}\,'
! m = (p_m, i_m, \rho_m, s_{b_m}, D_m, O_m, R_m)
from n-bound have n-bound\': n < length ts_{sb}\'
from m-bound have m-bound\': m < length ts_{sb}\' by (simp add: ts_{sb}\')
show (O_m \cup \text{all-acquired} s_{b_m}) \cap 
read-only-reads (acquired True (takeWhile (\text{Not} \circ \text{is-volatile-Write}_{sb}) s_{b_n}) O_n)
(dropWhile (\text{Not} \circ \text{is-volatile-Write}_{sb}) s_{b_n}) = 
\{
\}
proof (cases m=i)
  case True
  with neq-n-m have neq-n-i: n \neq i
  by auto
  with n-bound nth i-bound have nth\': ts_{sb}\n! n = (p_n, i_n, \rho_n, s_{b_n}, D_n, O_n, R_n)
  by (auto simp add: ts_{sb}\')
  note read-only-reads-unowned [OF n-bound\' i-bound neq-n-i nth\' ts_{sb}-i]
  moreover
  note A-no-read-only-reads [OF n-bound\' nth\']
  ultimately
  show \?thesis
    using True ts_{sb}-i neq-n-i nth m-bound\' m-bound\'
    by (auto simp add: ts_{sb}' O_{sb}' s_{b sb})
next
  case False
  note neq-m-i = this
  with m-bound mth i-bound have mth\': ts_{sb}\n! m = (p_m, i_m, \rho_m, s_{b_m}, D_m, O_m, R_m)
  by (auto simp add: ts_{sb}'
show ?thesis
proof (cases n=i)
  case True
  with ts_{sb}-i nth mth neq-m-i n-bound'
  show ?thesis
  by (auto simp add: ts_{sb}' s_{b sb}')
next
  case False
  with n-bound nth i-bound have nth\': ts_{sb}\n! n = (p_n, i_n, \rho_n, s_{b_n}, D_n, O_n, R_n)
  by (auto simp add: ts_{sb}')
  from read-only-reads-unowned [OF n-bound\' m-bound\' neq-n-m nth\' mth\'] False neq-m-i
  show ?thesis
by (clarsimp)
qed
qed
qed

next

show ownership-distinct \(ts_{sb}'\)

proof (unfold-locales)

fix \(i_1 \, j \, p_1 \, i_1 \, O_1 \, R_1 \, D_1 \, xs_1 \, sb_1 \, p_j \, is_j \, O_j \, R_j \, D_j \, xs_j \, sb_j\)

assume i-bound: \(i_1 < \text{length } ts_{sb}'\)
assume j-bound: \(j < \text{length } ts_{sb}'\)
assume i\(\neq j\):
assume ts-i: \(ts_{sb}'!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)
assume ts-j: \(ts_{sb}'!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)\)

show \((O_1 \cup \text{all-acquired } sb_1) \cap (O_j \cup \text{all-acquired } sb_j) = \{\}\)
proof (cases \(i_1 = i\))
case True
with \(i_1\neq j\)
  by simp

from i-bound \(ts_{i_1} ts_{sb}'\) True have sb1: \(sb_1 = []\)
  by (simp add: ts_{sb}' sb)
from j-bound have j-bound': \(j < \text{length } ts_{sb}\)
  by (simp add: ts_{sb}')

hence j-bound'': \(j < \text{length } (\text{map owned } ts_{sb})\)
  by simp
from ts-j i-j have ts-j': \(ts_{sb}'!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)\)
  by (simp add: ts_{sb}')

from A-unused-by-others [rule-format, OF - i-j] ts-j i-j j-bound'
have A \(\cap (O_j \cup \text{outstanding-refs is-volatile-Write}_{sb_j}) = \{\}\)
  by (auto simp add: Let-def ts_{sb}' owned-def)
moreover
from A-unacquired-by-others [rule-format, OF - i-j] ts-j i-j j-bound'
have A \(\cap \text{all-acquired } sb_j = \{\}\)
  by (auto simp add: Let-def ts_{sb}'

moreover
from ownership-distinct [OF i-bound j-bound' i-j ts_{sb}-i ts-j']
have O_{sb} \(\cap (O_j \cup \text{all-acquired } sb_j) = \{\}\)
  by (simp add: sb)
ultimately show ?thesis using ts-i1 True
  by (auto simp add: i-bound ts_{sb}' O_{sb}' sb' sb1)

next
case False
note i1-i = this
from i1-bound have i1-bound': \(i_1 < \text{length } ts_{sb}\)
  by (simp add: ts_{sb}')
hence i1-bound'': \(i_1 < \text{length } (\text{map owned } ts_{sb})\)
  by simp
from ts-i False have ts-i': \(ts_{sb}'!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)
  by (simp add: ts_{sb}')

579
show \( ? \text{thesis} \)
proof (cases \( j=i \))
case True
from A-unused-by-others [rule-format, OF - False [symmetric]] ts-i
False \( i_1 \text{-bound}' \)
  have \( A \cap (O_1 \cup \text{outstanding-refs} \text{is-volatile-Write}_{sb1}) = \{ \} \)
by (auto simp add: Let-def ts-sb \( \text{owned-def} \))
moreover
from A-unacquired-by-others [rule-format, OF - False [symmetric]] ts-i
False
\( i_1 \text{-bound}' \)
  have \( A \cap (O_1 \cup \text{all-acquired sb1}) = \{ \} \)
by (auto simp add: Let-def ts-sb \( \text{owned-def} \))
moreover
from ownership-distinct [OF \( i_1 \text{-bound}' \) \( i_1 \text{-bound} \)]
show \( (O_1 \cup \text{all-acquired sb1}) \cap O_{sb} = \{ \} \)
by (simp add: sb)
ultimately show \( ? \text{thesis} \)
using ts-j True
by (auto simp add: \( i_1 \text{-bound} \) \( \text{owned-def} \))
next
case False
from \( j \text{-bound} \)
  have \( j \text{-bound}' : j < \text{length ts}_{sb} \)
by (simp add: ts-sb \( \text{owned-def} \))
from ts-j False
  have ts-j': \( ts_{sb}'[j] = (p_j, \text{is}_j, \theta_j, sb_j, D_j, O_j, R_j) \)
by (simp add: ts-sb \( \text{owned-def} \))
from ownership-distinct [OF \( i_1 \text{-bound}' \) \( j \text{-bound}' \)]
show \( (O_1 \cup \text{all-acquired sb1}) \cap (O_j \cup \text{all-acquired sb}_j) = \{ \} \)
qed
qed
qed

have valid-hist': valid-history program-step ts-sb'
proof
from valid-history [OF \( i \text{-bound} \) ts-sb-\( i \)]
  have history-consistent (\( \theta_{sb}(t \mapsto \text{ret} (m_{sb} a) (f (\theta_{sb}(t \mapsto m_{sb} a)))) \) (hd-prog \( p_{sb} [\] [) by simp
from valid-history-nth-update [OF \( i \text{-bound} \) this]
show \( ? \text{thesis} \)
by (simp add: ts-sb' \( \theta_{sb}' \) \( sb' \) sb)
qed

from valid-reads [OF \( i \text{-bound} \) ts-sb-\( i \)]
  have reads-consist: reads-consistent False \( O_{sb} m_{sb} sb \).

  have valid-reads': valid-reads \( m_{sb}' \) ts-sb'
  proof (unfold-locales)
    fix \( j \) \( p_j \) \( i \) \( O_j \) \( R_j \) \( D_j \) \( \text{acq} \) \( \theta_j \) \( sb_j \)
    assume \( j \text{-bound} : j < \text{length ts}_{sb}' \)
    assume ts-j: \( ts_{sb}'[j] = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) \)
    show reads-consistent False \( O_j m_{sb}' sb_j \)
  end
proof (cases \( i=j \))

580
case True
from reads-consistent ts-j-bound sb show ?thesis
  by (clarsimp simp add: True m\sb\sb′ Write\sb\sb′ ts\sb′ \sb′)

next
case False
from j-bound have j-bound′: j < length ts\sb
  by (simp add: ts\sb′)
moreover from ts-j False have ts-j′: ts\sb′ ! j = (p_j, i_s, t_j, D_j, O_j, R_j)
  using j-bound by (simp add: ts\sb′)
ultimately have consis-m: reads-consistent False O_j m\sb\sb sb_j
  by (rule valid-reads)
let ?m′ = (m\sb\sb(a := f (θ\sb(t := m\sb\sb a))))
from a-unowned-others [rule-format, OF - False] j-bound′
obtain a-acq: a /∈ acquired True (takeWhile (Not o is-volatile-Write\sb) sb_j)
  and a-unsh: a /∈ all-shared (takeWhile (Not o is-volatile-Write\sb) sb_j)
  by auto
with a-notin-unforwarded-non-volatile-reads-drop [OF j-bound′ ts-j′ False]
have \forall a ∈ acquired True (takeWhile (Not o is-volatile-Write\sb) sb_j) O_j ∪ all-shared (takeWhile (Not o is-volatile-Write\sb) sb_j) ∪ unforwarded-non-volatile-reads (dropWhile (Not o is-volatile-Write\sb) sb_j) \{}.
?m′ a = m\sb\sb a
  by auto
from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop
  [where W = \{} simplified, OF this - - consis-m]
  acquired-reads-all-acquired′ [of (takeWhile (Not o is-volatile-Write\sb) sb_j) O_j]
  have reads-consistent False O_j (m\sb\sb(a := f (θ\sb(t := m\sb\sb a)))) sb_j
    by (auto simp del: fun-upd-apply)
thus ?thesis
  by (simp add: m\sb′)
qed

have valid-sharing′: valid-sharing (S\sb\sb ⊕ W R ⊕ A L) ts\sb′
proof (intro-locales)
show outstanding-non-volatile-writes-unshared (S\sb\sb ⊕ W R ⊕ A L) ts\sb′
proof (unfold-locales)
fix j p_j i_s \sb_j R_j D_j acq\sb_j x_s\sb_j sb_j
assume j-bound: j < length ts\sb′
assume jth: ts\sb′ ! j = (p_j, i_s, x_s, sb_j, D_j, O_j, R_j)
show non-volatile-writes-unshared (S\sb\sb ⊕ W R ⊕ A L) sb_j
proof (cases i=j)
  case True
  with i-bound jth show ?thesis
    by (simp add: ts\sb′ sb′ sb)
next
case False
from j-bound have j-bound′: j < length ts\sb
  by (auto simp add: ts\sb′)
from jth False have jth′: ts\sb′ ! j = (p_j, i_s, x_s, sb_j, D_j, O_j, R_j)
  by (auto simp add: ts\sb′)

581
from outstanding-non-volatile-writes-unshared [OF j-bound′ jth′]
have unshared: non-volatile-writes-unshared \( S_{sb} \),
have \( \forall a \in \text{dom} \ (S_{sb} \oplus W R \ominus A L) - \text{dom} S_{sb}, a \notin \text{outstanding-refs} \) is-non-volatile-Write_{sb} 

proof –
{  
fix a
assume a-in: \( a \in \text{dom} \ (S_{sb} \oplus W R \ominus A L) - \text{dom} S_{sb} \)
hence a-R: \( a \in R \)
by clarsimp
assume a-in-j: \( a \in \text{outstanding-refs} \) is-non-volatile-Write_{sb} \( sb_j \)
have False
proof –
from non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF outstanding-non-volatile-refs-owned-or-read-only [OF j-bound′ jth′]]
a-in-j
have \( a \in O_j \cup \text{all-acquired} \ sb_j \)
by auto
moreover
with ownership-distinct [OF i-bound j-bound′ jth′] a-R R-owned
show False
by blast
qed
}
thus ?thesis by blast
qed

from non-volatile-writes-unshared-no-outstanding-non-volatile-Write_{sb}
[OF unshared this]
show ?thesis .
qed
qed
next
show sharing-consis \((S_{sb} \oplus W R \ominus A L) ts_{sb}'\)
proof (unfold-locale)
fix \( j \) \( p_j \), is\( j \), \( O_j \), \( R_j \), \( D_j \), acq \( x_j \), \( sb_j \)
assume j-bound: \( j < \text{length} \ ts_{sb}' \)
assume jth: \( ts_{sb}'! j = (p_j, is\( j \), x\( j \), sb\( j \), D\( j \), O\( j \), R\( j \)) \)
show sharing-consistent \((S_{sb} \oplus W R \ominus A L) O_j sb_j \)
proof (cases i=j)
case True
with i-bound jth show ?thesis
by (simp add: ts_{sb}' sb' sb)
next
case False
from j-bound have j-bound': \( j < \text{length} \ ts_{sb} \)
by (auto simp add: ts_{sb}' )
from jth False have jth': \( ts_{sb} ! j = (p_j, is\( j \), x\( j \), sb\( j \), D\( j \), O\( j \), R\( j \)) \)
by (auto simp add: ts_{sb}' )
from sharing-consis [OF j-bound′ jth′]

582
have consis: sharing-consistent $S_{sb} \mathcal{O}_j sb_j$.

have acq-cond: all-acquired $sb_j \cap \text{dom } S_{sb} - \text{dom } (S_{sb} \oplus W R \ominus A L) = \{\}$
proof -
{ fix a
assume a-acq: $a \in$ all-acquired $sb_j$
assume $a \in$ dom $S_{sb}$
assume a-L: $a \in L$
have False
proof -
from A-unacquired-by-others [rule-format, of j,OF - False] j-bound' jth'
have $A \cap$ all-acquired $sb_j = \{\}$
  by auto
with a-acq a-L L-subset
show False
  by blast
qed
}
  thus ?thesis
by auto
qed

have uns-cond: all-unshared $sb_j \cap \text{dom } (S_{sb} \oplus W R \ominus A L) - \text{dom } S_{sb} = \{\}$
proof -
{ fix a
assume a-uns: $a \in$ all-unshared $sb_j$
assume $a \notin L$
assume a-R: $a \in R$
have False
proof -
from unshared-acquired-or-owned [OF consis] a-uns
have $a \in$ all-acquired $sb_j \cup \mathcal{O}_j$ by auto
with ownership-distinct [OF i-bound j-bound' False $ts_{sb}$-i jth'] R-owned a-R
show False
  by blast
qed
}
  thus ?thesis
by auto
qed

from sharing-consistent-preservation [OF consis acq-cond uns-cond]
show ?thesis
  by (simp add: $ts_{sb}'$
qed

qed

next
show unowned-shared $(S_{sb} \oplus W R \ominus A L) \ ts_{sb}'$
proof (unfold-locals)
\[ \text{show} - \bigcup (\{\lambda (-, -, -, O, -) \setminus \text{set ts}\}) \subseteq \text{dom} (\mathcal{S}_{sb} \oplus_W R \ominus_A L) \]

\text{proof} –

\text{have} s: \bigcup (\{\lambda (-, -, -, O, -) \setminus \text{set ts}\}) = \bigcup (\{\lambda (-, -, -, O, -) \setminus \text{set ts}\}) \cup A - R

\text{apply (unfold ts'O)}
\text{apply (rule acquire-release-ownership-nth-update [OF R-owned i-bound ts_i])}
\text{apply fact}
\text{done}

\text{note} unowned-shared L-subset A-R
\text{then}
\text{show ?thesis}
\quad \text{apply (simp only: s)}
\quad \text{apply auto}
\text{done}
\text{qed}
\text{next}
\text{show read-only-unowned (\mathcal{S}_{sb} \oplus_W R \ominus_A L) ts_{sb}'}
\text{proof}
\quad \text{fix} j \ p_j \ is_j \ O_j \ R_j \ D_j \ acq_j \ xs_j \ sb_j
\quad \text{assume} j-bound: j < \text{length ts}_{sb}'
\quad \text{assume jth: ts}_{sb}' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
\quad \text{show O_j \cap read-only (\mathcal{S}_{sb} \oplus_W R \ominus_A L) = \{\}}
\quad \text{proof (cases i=j)}
\quad \quad \text{case True}
\quad \quad \quad \text{from read-only-unowned [OF i-bound ts_{sb}-i] R-owned A-R}
\quad \quad \quad \text{have (O}_{sb} \cup A - R) \cap read-only (\mathcal{S}_{sb} \oplus_W R \ominus_A L) = \{\}
\quad \quad \quad \quad \text{by (auto simp add: in-read-only-convs )}
\quad \quad \quad \text{with ts_{sb}-i i-bound True}
\quad \quad \quad \text{show ?thesis}
\quad \quad \quad \quad \text{by (auto simp add: O_{sb}' ts_{sb}')}
\quad \quad \text{next}
\quad \quad \text{case False}
\quad \quad \quad \text{from j-bound have j-bound': j < \text{length ts}_{sb}}
\quad \quad \quad \quad \text{by (auto simp add: ts_{sb}' )}
\quad \quad \quad \text{with False jth have jth': ts}_{sb}' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
\quad \quad \quad \quad \text{by (auto simp add: ts_{sb}' )}
\quad \quad \text{from read-only-unowned [OF j-bound' jth']}
\quad \quad \text{have O_j \cap read-only \mathcal{S}_{sb} = \{\}.
\quad \quad \text{moreover}
\quad \quad \text{from A-unowned-by-others [rule-format, OF - False] j-bound' jth'}
\quad \quad \quad \text{have A \cap O_j = \{\}
\quad \quad \quad \quad \text{by (auto dest: all-shared-acquired-in )}
\quad \quad \quad \text{moreover}
\quad \quad \quad \text{from ownership-distinct [OF i-bound j-bound' False ts_{sb}-i jth']}
\quad \quad \quad \text{have O_{sb} \cap O_j = \{\}
\quad \quad \quad \quad \text{by auto

584
moreover note R-owned A-R
ultimately show ?thesis
  by (fastforce simp add: in-read-only-convs split: if-split-asm)
qed

next
show no-outstanding-write-to-read-only-memory \(S_{sb} \oplus W R \ominus A L \) ts_{sb} \'
proof
  fix j p_j is_j O_j D_j acq_j x_{sj} s_{bj}
  assume j-bound: \(j < \) length ts_{sb} \'
  assume jth: \(ts_{sb} \' \_j = (p_j, is_j, x_{sj}, s_{bj}, D_j, O_j, R_j)\)
show no-write-to-read-only-memory \((S_{sb} \oplus W R \ominus A L) \) sb_j
proof (cases i=j)
case True
  with jth ts_{sb} -i i-bound
  show ?thesis
  by (auto simp add: sb sb \_\_ts_{sb} \_\_)
next
case False
from j-bound have j-bound\': \(j < \) length ts_{sb} 
  by (auto simp add: ts_{sb} \_\_)
with False jth have jth\': \(ts_{sb} \_j = (p_j, is_j, x_{sj}, s_{bj}, D_j, O_j, R_j)\)
  by (auto simp add: ts_{sb} \_\_)
from no-outstanding-write-to-read-only-memory [OF j-bound\' jth\']
have nw: no-write-to-read-only-memory \(S_{sb} \) sb_{j}.
have R \cap outstanding-refs is-Write_{sb} sb_{j} = \{\}
proof
  note dist = ownership-distinct [OF i-bound j-bound\' False ts_{sb}-i jth\']
  from non-volatile-owned-or-read-only-outstanding-non-volatile-writes
  [OF outstanding-non-volatile-refs-owned-or-read-only [OF j-bound\' jth\']]
dist
  have outstanding-refs is-non-volatile-Write_{sb} sb_{j} \cap O_{sb} = \{\}
by auto
  moreover
  from outstanding-volatile-writes-unowned-by-others [OF j-bound\' i-bound]
  False [symmetric] jth\' ts_{sb}-i ]
  have outstanding-refs is-volatile-Write_{sb} sb_{j} \cap O_{sb} = \{\}
by auto
  ultimately have outstanding-refs is-Write_{sb} sb_{j} \cap O_{sb} = \{\}
by (auto simp add: misc-outstanding-refs-convs)
  with R-owned
  show ?thesis by blast
qed

then
have \(\forall a \in outstanding-refs is-Write_{sb} sb_{j}.
  a \in read-only \((S_{sb} \oplus W R \ominus A L) \rightarrow a \in read-only S_{sb}\)
by (auto simp add: in-read-only-convs)
from no-write-to-read-only-memory-read-only-reads-eq [OF nw this]
show ?thesis .

585
have tmps-distinct': tmps-distinct ts_{sb}'
proof (intro-locales)
from load-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-load-tmps is_{sb}'
  by (auto simp add: is_{sb} split: instr.splits)
from load-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-distinct ts_{sb}' by (simp add: ts_{sb}' sb'O_{sb}' is_{sb})
  next
from read-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-read-tmps [] by (simp add: ts_{sb}' sb'O_{sb}')
from read-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct ts_{sb}' by (simp add: ts_{sb}' sb'O_{sb}')
next
from load-tmps-read-tmps-distinct [OF i-bound ts_{sb}-i]
  load-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' ∩ read-tmps [] = {}
  by (clarsimp)
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct ts_{sb}' by (simp add: ts_{sb}' sb'O_{sb}')
qed

have valid-sops': valid-sops ts_{sb}'
proof -
from valid-store-sops [OF i-bound ts_{sb}-i]
obtain valid-store-sops': ∀ sop∈store-sops is_{sb}' . valid-sop sop
  by (auto simp add: is_{sb} ts_{sb}' sb'O_{sb}')
from valid-sops-nth-update [OF i-bound valid-sops', where sb= []]
show ?thesis by (auto simp add: ts_{sb}' sb'O_{sb}')
qed

have valid-dd': valid-data-dependency ts_{sb}'
proof -
from data-dependency-consistent-instrs [OF i-bound ts_{sb}-i]
obtain dd-is: data-dependency-consistent-instrs (dom θ_{sb}') is_{sb}'
  by (auto simp add: is_{sb} θ_{sb}')
from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' ∩ (fst' write-sops []) = {}
  by (auto simp add: write-sops-append)
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show ?thesis by (simp add: ts_{sb}' sb'O_{sb}')
qed

have load-tmps-fresh': load-tmps-fresh ts_{sb}'
proof  
  from load-tmps-fresh [OF i-bound ts\sb-i]  
  have load-tmps (RMW a t (D,f) cond ret A L R W # is\sb') \cap dom \vartheta\sb = {}  
    by (simp add: is\sb)  
  moreover  
  from load-tmps-distinct [OF i-bound ts\sb-i] have t \notin load-tmps is\sb  
    by (auto simp add: is\sb)  
  ultimately have load-tmps is\sb' \cap dom (\vartheta\sb(t \mapsto \text{ret (m\sb a) (f (\vartheta\sb(t \mapsto m\sb a)))})) = {}  
    by auto  
  from load-tmps-fresh-nth-update [OF i-bound this]  
  show ?thesis by (simp add: ts\sb sb sb')  
  qed  
  
  from enough-flushs-nth-update [OF i-bound, where sb=[]]  
  have enough-flushs\sb': enough-flushs ts\sb sb'  
    by (auto simp: ts\sb sb sb')  
  
  have valid-program-history\sb': valid-program-history ts\sb sb'  
  proof  
    have causal\sb': causal-program-history is\sb sb'  
      by (simp add: is\sb sb sb')  
    have last-prog p\sb sb' = p\sb  
      by (simp add: sb sb sb')  
    from valid-program-history-nth-update [OF i-bound causal\sb']  
    show ?thesis by (simp add: ts\sb sb sb')  
  qed  
  
  from is-sim have is: is = RMW a t (D,f) cond ret A L R W # is\sb'  
    by (simp add: suspends sb is\sb)  
  from direct-memop-step.RMWWrite [where cond=cond and \vartheta=\vartheta\sb and m=m, OF cond]  
  have (RMW a t (D,f) cond ret A L R W # is\sb', \vartheta\sb, (),m, D, O\sb,R\sb, S) \Rightarrow  
    (is\sb',\vartheta\sb(t \mapsto \text{ret (m\sb a) (f (\vartheta\sb(t \mapsto m\sb a)))}), (),  
     m(a := f (\vartheta\sb(t \mapsto m a))), False, O\sb \cup A - R, Map.empty, S \oplus W R \ominus A L).  
  from direct-computation.concurrent-step.Memop [OF i-bound\sb' ts-i this]  
  have (ts, m, S) \Rightarrow_d (ts[i := (p\sb, is\sb',\vartheta\sb(t \mapsto \text{ret (m a) (f (\vartheta\sb(t \mapsto m a)))}), ()), False,  
     O\sb \cup A - R, Map.empty]),  
    m(a := f (\vartheta\sb(t \mapsto m a))), S \oplus W R \ominus A L).  
  moreover  
  have tmps-commute: \vartheta\sb(t \mapsto \text{ret (m\sb a) (f (\vartheta\sb(t \mapsto m\sb a))))) =  
    (\vartheta\sb | (\text{dom } \vartheta\sb - \{t\}))(t \mapsto \text{ret (m\sb a) (f (\vartheta\sb(t \mapsto m\sb a)))))  
    apply (rule ext)  
    apply (auto simp add: restrict-map-def domIff)  
  done  

from a-unflushed \( t_{sb\cdot i} \)

have a-unflushed':

\[ \forall j < \text{length } t_{sb} \] 

(\text{let } (\cdot,\cdot,\cdot, t_{sb\cdot j},\cdot,\cdot,\cdot) = t_{sb\cdot j}! \text{ in } a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}) )

by auto

have all-shared-L: \( \forall i \) \( p \) is \( \mathcal{O} \mathcal{R} \mathcal{D} \) acq \( \partial \) sb. \( i < \text{length } t_{sb} \rightarrow t_{sb} ! i = (p, \text{is}, \partial, \text{sb}, D, \mathcal{O}, \mathcal{R}) \rightarrow \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}) \cap L = \{\}

proof –

\[
\{ \\
\text{fix } j \text{ p} j \text{ is} j \mathcal{O} j \mathcal{R} j D j \partial j \text{ sb} j x \\
\text{assume j-bound: } j < \text{length } t_{sb} \\
\text{assume jth: } t_{sb\cdot j}! i = (p, \text{is}, \partial, \text{sb}, D, \mathcal{O}, \mathcal{R}) \\
\text{assume x-shared: } x \in \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}) \\
\text{assume x-L: } x \in L \\
\text{have False} \\
\text{proof (cases } i=j) \\
\text{case True with x-shared } t_{sb\cdot i} \text{ jth show False by (simp add: sb)} \\
\text{next} \\
\text{case False} \\
\text{show False} \\
\text{proof –} \\
\text{ from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]} \\
\text{ have all-shared sb} j \subseteq \text{all-acquired sb} j \cup \mathcal{O} j, \\
\text{moreover have all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}) \subseteq \text{all-shared sb} j \\
\text{using all-shared-append [of (takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}) \\
(\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})] \\
\text{by auto} \\
\text{ moreover} \\
\text{ from A-unacquired-by-others [rule-format, OF - False] jth j-bound} \\
\text{ have A } \cap \text{ all-acquired sb} j = \{\} \text{ by auto} \\
\text{ moreover} \\
\text{ from A-unowned-by-others [rule-format, OF - False] jth j-bound} \\
\text{ have A } \cap \mathcal{O} j = \{\} \text{ by (auto dest: all-shared-acquired-in)} \\
\text{ ultimately} \\
\text{ show False} \\
\text{ using L-subset x-L x-shared} \\
\text{ by blast} \\
\text{ qed} \\
\text{ qed} \\
\text{588}
have all-shared-A: \(\forall i\ p \in \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \emptyset \ \text{sb}. \ i < \text{length ts}_{\text{sb}} \rightarrow\)
\[\text{ts}_{\text{sb}} ! i = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow\]
\[\text{all-shared (takeWhile (Not o is-volatile-Write}_{\text{sb}}) \text{sb}) \cap A = \{\}\]

proof –

\{ fix j p_j \in_j \mathcal{O}_j \ \mathcal{R}_j \ \mathcal{D}_j \ \emptyset_j \ \text{sb}_j x \ assumption j-bound: j < \text{length ts}_{\text{sb}} assumption jth: ts_{\text{sb}}!j = (p_j, \text{is}_j, \emptyset_j, \text{sb}_j, \mathcal{D}_j, \mathcal{O}_j, \mathcal{R}_j) assumption x-shared: x \in \text{all-shared (takeWhile (Not o is-volatile-Write}_{\text{sb}}) \text{sb}_j) assumption x-A: x \in A have False proof (cases i=j) case True with x-shared ts_{\text{sb}-i} jth show False by (simp add: sb) next case False show False proof – from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]] have all-shared \text{sb}_j \subseteq \text{all-acquired \text{sb}_j} \cup O_j. moreover have all-shared (takeWhile (Not o is-volatile-Write}_{\text{sb}}) \text{sb}_j) \subseteq \text{all-shared \text{sb}_j} using all-shared-append [of (takeWhile (Not o is-volatile-Write}_{\text{sb}}) \text{sb}_j) (dropWhile (Not o is-volatile-Write}_{\text{sb}}) \text{sb}_j)] by auto moreover from A-unacquired-by-others [rule-format, OF - False] jth j-bound have A \cap \text{all-acquired \text{sb}_j} = \{\} by auto moreover from A-unowned-by-others [rule-format, OF - False] jth j-bound have A \cap O_j = \{\} by (auto dest: all-shared-acquired-in)

ultimately show False using x-A x-shared by blast qed qed qed

thus \(\approx\)thesis by blast qed

hence all-shared-L: \(\forall i\ p \in \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \emptyset \ \text{sb}. \ i < \text{length ts}_{\text{sb}} \rightarrow\)
\[\text{ts}_{\text{sb}} ! i = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow\]
all-shared \((\text{takeWhile} \circ \text{is-volatile-Write}_{sb}) \cap L = \{\}\)

**using** \(L\)-subset by blast

**have** all-unshared-R: \(\forall i \; p \; \text{is} \; R \; D \; \varnothing \; sb.\; i < \text{length} \; ts_{sb} \rightarrow \)
\(ts_{sb}!i = (p, \text{is}, \varnothing, sb, D, O, R) \rightarrow \)
all-unshared \((\text{takeWhile} \circ \text{is-volatile-Write}_{sb}) \cap R = \{\}\)

**proof** – 
\[
\begin{align*}
\{ & \text{fix} \; j \; p_j \; is_j \; O_j \; R_j \; D_j \; \varnothing \; sb_j \; x \\
& \text{assume} \; j\text{-bound: } j < \text{length} \; ts_{sb} \\
& \text{assume} \; j\text{th: } ts_{sb}!j = (p_j, is_j, \varnothing, sb_j, D_j, O_j, R_j) \\
& \text{assume} \; x\text{-unshared: } x \in \text{all-unshared} \; (\text{takeWhile} \circ \text{is-volatile-Write}_{sb}) \; sb_j \\
& \text{assume} \; x\text{-R: } x \in R \\
& \text{have False} \\
\text{proof} \quad \text{(cases } i=j) \\
\text{case} \; \text{True with } x\text{-unshared } ts_{sb}-i \; j\text{th} \text{ show False by (simp add: sb)} \\
\text{next} \\
\text{case} \; \text{False} \\
\text{show False} \\
\text{proof} \quad \text{from unshared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]} \\
& \text{have all-unshared } sb_j \subseteq \text{all-acquired } sb_j \cup O_j, \\
\text{moreover have} \; \text{all-unshared} \; (\text{takeWhile} \circ \text{is-volatile-Write}_{sb}) \; sb_j \subseteq \text{all-unshared} \; sb_j \\
\text{using} \; \text{all-unshared-append [of (takeWhile} \circ \text{is-volatile-Write}_{sb}) \; sb_j) \\
& (\text{dropWhile} \circ \text{is-volatile-Write}_{sb}) \; sb_j] \\
\text{by auto} \\
\text{moreover} \\
\text{note ownership-distinct [OF i-bound j-bound False } ts_{sb}-i \text{ jth]} \\
\text{ultimately} \\
\text{show False} \\
\text{using } R\text{-owned } x\text{-R } x\text{-unshared} \\
\text{by blast} \\
\text{qed} \\
\text{qed} \\
\}
\]

**thus** \(\text{thesis by blast} \)

**have** all-acquired-R: \(\forall i \; p \; \text{is} \; O \; R \; D \; \varnothing \; sb.\; i < \text{length} \; ts_{sb} \rightarrow \)
\(ts_{sb}!i = (p, \text{is}, \varnothing, sb, D, O, R) \rightarrow \)
all-acquired \((\text{takeWhile} \circ \text{is-volatile-Write}_{sb}) \cap R = \{\}\)

**proof** – 
\[
\begin{align*}
\{ & \text{fix} \; j \; p_j \; is_j \; O_j \; R_j \; D_j \; \varnothing \; sb_j \; x \\
& \text{assume} \; j\text{-bound: } j < \text{length} \; ts_{sb} \\
& \text{assume} \; j\text{th: } ts_{sb}!j = (p_j, is_j, \varnothing, sb_j, D_j, O_j, R_j) \\
\}
\]

590
assume \( x \)-acq: \( x \in \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j) \)

assume \( x \)-R: \( x \in R \)

have False

proof (cases \( i=j \))

case True with \( x \)-acq \( ts_{sb} \)-i jth show False by (simp add: sb)

next

case False

show False proof

proof –

from \( x \)-acq have \( x \in \text{all-acquired} \hspace{1px} sb_j \)

using all-acquired-append [of \( \text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j \)

\( \text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j \)]

by auto

moreover

note ownership-distinct [OF \( i \)-bound \( j \)-bound False ts_{sb}-i jth]

ultimately

show False

using \( R \)-owned \( x \)-R

by blast

qed

qed

} thus \( \theta \)thesis by blast

qed

have all-shared-R: \( \forall \hspace{1px} i \hspace{1px} p \hspace{1px} \text{O} \hspace{1px} \text{R} \hspace{1px} D \hspace{1px} \emptyset \hspace{1px} \text{sb}. \hspace{1px} i < \text{length} \hspace{1px} ts_{sb} \rightarrow \)

\( ts_{sb} \) ! \( i \) = \( (p, \text{is}, \emptyset, \text{sb}, D, O, R) \rightarrow \)

all-shared (\( \text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb \)) \( \cap \hspace{1px} R = \{\} \)

proof –

\{ fix \( j \hspace{1px} p_j \hspace{1px} \text{is} \hspace{1px} \text{O} \hspace{1px} \text{R} \hspace{1px} D_j \hspace{1px} \emptyset j \hspace{1px} \text{sb}_j \hspace{1px} x \)

assume \( j \)-bound: \( j < \text{length} \hspace{1px} ts_{sb} \)

assume \( j \)-th: \( ts_{sb}[j] = (p_j, \text{is}_j, \emptyset j, \text{sb}_j, D_j, O_j, R_j) \)

assume \( x \)-shared: \( x \in \text{all-shared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j) \)

assume \( x \)-R: \( x \in R \)

have False

proof (cases \( i=j \))

case True with \( x \)-shared \( ts_{sb} \)-i jth show False by (simp add: sb)

next

case False

show False proof

proof –

from all-shared-acquired-or-owned [OF sharing-consis [OF \( j \)-bound \( j \)-th]]

have all-shared \( sb_j \subseteq \text{all-acquired} \hspace{1px} \text{sb}_j \cup O_j. \)

moreover have all-shared (\( \text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j \)) \( \subseteq \text{all-shared} \hspace{1px} sb_j \)

using all-shared-append [of (\( \text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j \))

(\( \text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \hspace{1px} sb_j) \)]

591
by auto

moreover

note ownership-distinct [OF i-bound j-bound False ts\_sb\_i jth]

ultimately

show False

using R-owned x-R x-shared

by blast

qed

qed

thus ?thesis by blast

qed

from share-all-until-volatile-write-commute [OF (ownership-distinct ts\_sb)]

(sharing-consis S\_sb ts\_sb)

all-shared-L all-shared-A all-acquired-R all-unshared-R all-shared-R

have share-commute: share-all-until-volatile-write ts\_sb S\_sb \oplus_W R \ominus_A L =

share-all-until-volatile-write ts\_sb (S\_sb \oplus_W R \ominus_A L).

\{

fix j p\_j is\_j O\_j \ R\_j \ D\_j \ \theta\_j \ sb\_j \ x

assume jth: ts\_sb\_j = (p\_j, is\_j, \theta\_j, sb\_j, D\_j, O\_j, R\_j)

assume j-bound: j < length ts\_sb

assume neq: i \neq j

have release (takeWhile (Not \circ is-volatile-Write\_sb) sb\_j)

(dom S\_sb \cup R \ominus L) R\_j

= release (takeWhile (Not \circ is-volatile-Write\_sb) sb\_j)

(dom S\_sb) R\_j

proof –

\{

fix a

assume a-in: a \in all-shared (takeWhile (Not \circ is-volatile-Write\_sb) sb\_j)

have (a \in (dom S\_sb \cup R \ominus L)) = (a \in dom S\_sb)

proof –

from A-unowned-by-others [rule-format, OF j-bound neq ] jth
A-unacquired-by-others [rule-format, OF - neq] j-bound

have A-dist: A \cap (O\_j \cup all-acquired sb\_j) = \{

by (auto dest: all-shared-acquired-in)

from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]] a-in
all-shared-append [of (takeWhile (Not \circ is-volatile-Write\_sb) sb\_j)]

(dropWhile (Not \circ is-volatile-Write\_sb) sb\_j)]

have a-in: a \in O\_j \cup all-acquired sb\_j

by auto

with ownership-distinct [OF i-bound j-bound neq ts\_sb\_i jth]

have a \notin (O\_sb \cup all-acquired sb) by auto

with A-dist R-owned A-R A-shared-owned L-subset a-in

592
obtain \( a \notin R \) and \( a \notin L \)
by fastforce
then show \(?\)thesis by auto
qed

} then
show \(?\)thesis
apply –
apply (rule release-all-shared-exchange)
apply auto
done
qed

note release-commute = this

have \((ts_{sb}, m_{sb} (a := f (\bar{\theta}_{sb} (t \mapsto m_{sb} a)))) , S_{sb} \) \sim (ts[i := (p_{sb} , i_{sb} , \theta_{sb} (t \mapsto \text{ret} (m a) (f (\bar{\theta}_{sb} (t \mapsto m a)))) , () , False , O_{sb} \cup A - R , Map.empty)] , m(a := f (\bar{\theta}_{sb} (t \mapsto m a)))) , S \oplus W R \ominus A L\)
apply (rule sim-config, intros)
apply (simp only: m-a )
apply (simp only: m )
apply (simp only: flush-all-until-volatile-write-update-other [OF a-unflushed', symmetric] ts_{sb}')
apply (simp add: flush-all-until-volatile-nth-update-unused [OF i-bound ts_{sb}-i, simplified sb] sb')
apply (simp add: ts_{sb}' sb' O_{sb}' m
flush-all-until-volatile-nth-update-unused [OF i-bound ts_{sb}-i, simplified sb])
using share-all-until-volatile-write-RMW-commute [OF i-bound ts_{sb}-i [simplified is_{sb} sb]]
apply (clarsimp simp add: S ts_{sb}' S_{sb}' i_{sb} S_{sb}' O_{sb}' R_{sb}' \theta_{sb}' sb' sb share-commute)
using leq
apply (simp add: ts_{sb}')
using i-bound i-bound' ts-sim
apply (clarsimp simp add: Let-def nth-list-update
 ts_{sb}' sb' O_{sb}' R_{sb}' S_{sb}' \theta_{sb}' D_{sb}' ex-not m-a
split: if-split-asm)
apply (rule conjI)
apply clarsimp
apply (rule tmps-commute)
apply clarsimp
apply (frule (2) release-commute)
apply clarsimp
apply fastforce
done
ultimately
show \(?\)thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-sops'
valid-dd' load-tmps-fresh' enough-flushs'
valid-program-history' valid' m_{sb}' S_{sb}'
by (auto simp del: fun-upd-apply)
next
case (SBHGhost A L R W)
then obtain

\( i_{sb} : i_{sb} = \text{Ghost A L R W} \# i_{sb}' \) and

\( O_{sb} : O_{sb} = O_{sb} \) and

\( R_{sb} : R_{sb} = R_{sb} \) and

\( \theta_{sb} : \theta_{sb} = \theta_{sb} \) and

\( D_{sb} : D_{sb} = D_{sb} \) and

\( m_{sb} : m_{sb} = m_{sb} \) and

\( S_{sb} : S_{sb} = S_{sb} \)

by auto

\[\begin{align*}
\text{from & safe-memop-flush-sb [simplified is_{sb}] obtai} & \text{n} \\
\text{L-subset: } & L \subseteq A \text{ and} \\
A \text{-shared-owned: } & A \subseteq \text{dom (share ?drop-sb S)} \cup \text{acquired True sb } O_{sb} \text{ and} \\
R \text{-acq: } & R \subseteq \text{acquired True sb } O_{sb} \text{ and} \\
A \text{-R: } & A \cap R = \{\} \text{ and} \\
A \text{-unowned-by-others-ts: } & \forall j < \text{length (map owned ts)}. \ i \neq j \rightarrow (A \cap \text{(owned (ts!j) } \cup \text{dom (released (ts!j)))) = \{\}) \\
\text{by & cases auto} \\
\text{from & A-unowned-by-others-ts ts-sim leq} \\
\text{have & A-unowned-by-others: } \\
\forall j < \text{length ts}_{sb}. \ i \neq j \rightarrow (\text{let } (-, -, sb_j, -, \text{O}_{j-}) = ts_{sb}!j) \\
in A \cap \text{(acquired True (takeWhile (Not is-volatile-Write_{sb}) sb_j)) } \text{O}_j \cup \\
\text{all-shared (takeWhile (Not is-volatile-Write_{sb}) sb_j)) = \{\}) \\
\text{apply (clarsimp simp add: Let-def)} \\
\text{subgoal for } j \\
\text{apply (drule-tac x=j in spec)} \\
\text{apply (force simp add: dom-release-takeWhile)} \\
\text{done} \\
\text{done} \\
\text{have & A-unused-by-others: } \\
\forall j < \text{length (map O}_{sb} \text{ ts}_{sb}). \ i \neq j \rightarrow \\
(\text{let } (O_j, sb_j) = \text{map O}_{sb} \text{ ts}_{sb}! j) \\
in A \cap \text{outstanding-refs is-volatile-Write_{sb} sb_j} = \{\}) \\
\text{proof –} \\
\{ \\
\text{fix } j \ O_{j} sb_j \\
\text{assume & j-bound: } j < \text{length (map owned ts}_{sb}) \\
\text{assume & neq-i-j: } i \not\neq j \\
\text{assume & ts}_{sb-j}: (\text{map O}_{sb} ts_{sb})!j = (O_j, sb_j) \\
\text{assume & conflict: } A \cap \text{outstanding-refs is-volatile-Write_{sb} sb_j} \not= \{\} \\
\text{have & False} \\
\text{proof –} \\
\text{from & j-bound leq} \\
\text{have & j-bound': } j < \text{length (map owned ts)} \text{ by auto} \\
\text{from & j-bound have & j-bound'': } j < \text{length ts}_{sb} \text{ by auto} \\
\text{from & j-bound' have & j-bound'': } j < \text{length ts} \text{ by auto} \\
\end{align*}\]
from conflict obtain a' where
  a'-in: a' ∈ A and
  a'-in-j: a' ∈ outstanding-refs is-volatile-Write_{sb} sb_j

by auto

let ?take-sb_j = (takeWhile (Not is-volatile-Write sb) sb_j)
let ?drop-sb_j = (dropWhile (Not is-volatile-Write sb) sb_j)

from ts-sim [rule-format, OF j-bound''] ts_{sb-j} j-bound
obtain p_j suspends_j is_{sb_j} D_{sb_j} R_j i_{sb_j} where
  ts_{sb-j} ! j = (p_j, i_{sb_j}, D_{sb_j}, O_j, R_j) and
  suspends_j = ?drop-sb_j and
  D_j: D_{sb_j} = (D_j ∨ outstanding-refs is-volatile-Write_{sb} sb_j ≠ {}) and
  i_{sb_j}: instrs suspends_j @ i_{sb_j} = i_{sb_j} @ prog-instrs suspends_j and
  ts_j: ts_{sb-j} = (hd-prog p_j suspends_j, i_{sb_j},
  \varnothing_{sb_j} |= (dom \varnothing_{sb_j} − read-tmps suspends_j), ()
  D_{sb_j}, acquired True \take-sb_j O_j, release \take-sb_j (dom S_{sb}) R_j)
apply (cases ts_{sb-j})
apply (force simp add: Let-def)
done

have a' ∈ outstanding-refs is-volatile-Write_{sb} suspends_j
proof −
  from a'-in-j
  have a' ∈ outstanding-refs is-volatile-Write_{sb} (?take-sb_j @ ?drop-sb_j)
by simp
  thus ?thesis
apply (simp only: outstanding-refs-append suspends_j)
apply (auto simp add: outstanding-refs-conv dest: set-takeWhileD)
done

from split-volatile-Write_{sb-in-outstanding-refs} [OF this]
obtain sop v ys zs A' L' R' W' where
  split-suspends_j: suspends_j = ys @ Write_{sb} True a' sop v A' L' R' W' # zs (is suspends_j = ?suspends)
by blast

from direct-memop-step.Ghost [where \varnothing=\varnothing_{sb} and m=flush ?drop-sb m]
have (Ghost A L R W# i_{sb}',
  \varnothing_{sb}, (), flush ?drop-sb m, D_{sb},
  acquired True sb O_{sb}, release sb (dom S_{sb}) R_{sb}, share ?drop-sb S) →
(i_{sb}', \varnothing_{sb}, (), flush ?drop-sb m, D_{sb},
  acquired True sb O_{sb} ∪ A − R,
  augment-rels (dom (share ?drop-sb S)) R (release sb (dom S_{sb}) R_{sb}),
  share ?drop-sb S @ W R @ A L).

from direct-computation.concurrent-step.Memop [OF
i-bound-ts [simplified is$_{sb}$] ts'i [simplified is$_{sb}$] this [simplified is$_{sb}$]

**have** store-step: (?ts', flush ?drop-sb m, share ?drop-sb $S$) ⇒

(?ts'' : = (?sb: is$_{sb}$', $\theta$$_{sb}$: ((),$D$$_{sb}$)), acquired True sb $O$$_{sb}$ $A$ - $R$, augment-rels

(dom (share ?drop-sb $S$)) $R$ (release sb (dom $S$$_{sb}$) $R$$_{sb}$)),

flush ?drop-sb m, share ?drop-sb $S$$\oplus$$W$$R$$\ominus$$A$$L$)

(is - ⇒ $d$ (?ts-A, ?m-A, ?share-A))

by (simp add: is$_{sb}$)

**from** i-bound' **have** i-bound'': i < length ?ts-A

by simp

**from** valid-program-history [OF j-bound'' ts$_{sb}$-j]
**have** causal-program-history is$_{sbj}$ sb$_j$.

then **have** cph: causal-program-history is$_{sbj}$ ?suspends

apply -
apply (rule causal-program-history-suffix [where sb=?take-sb$_j$ ])
apply (simp only: split-suspends$_j$ [symmetric] suspends$_j$)
apply (simp add: split-suspends$_j$)

**done**

**from** ts$_j$ neq-i-j j-bound
**have** ts-A-j: ?ts-A!j = (hd-prog p$_j$ (ys @ Write$_{sb}$ True a' sop v A' L' R' W' # zs), is$_j$,

$\theta$$_{sbj}$ | (dom $\theta$$_{sbj}$ - read-tmps (ys @ Write$_{sb}$ True a' sop v A' L' R' W' # zs)), ((), $D$$_j$),

acquired True ?take-sb$_j$ $O$$_j$, release ?take-sb$_j$ (dom $S$$_{sb}$) $R$$_{ij}$)

by (simp add: split-suspends$_j$)

**from** j-bound'' i-bound' neq-i-j **have** j-bound'''': j < length ?ts-A

by simp

**from** valid-last-prog [OF j-bound'' ts$_{sb}$-j] **have** last-prog: last-prog p$_j$ sb$_j$ = p$_j$.

then **have** lp: last-prog p$_j$ ?suspends = p$_j$

apply -
apply (rule last-prog-same-append [where sb=?take-sb$_j$ ])
apply (simp only: split-suspends$_j$ [symmetric] suspends$_j$)
apply simp

**done**

**from** valid-reads [OF j-bound'' ts$_{sb}$-j]
**have** reads-consis: reads-consistent False $O$$_j$ m$_{sb}$ sb$_j$.

**from** reads-consistent-flush-all-until-volatile-write [OF (valid-ownership-and-sharing $S$$_{sb}$ ts$_{sb}$') j-bound'']

**have** reads-consis-m: reads-consistent True (acquired True ?take-sb$_j$ $O$$_j$) m suspends$_j$

by (simp add: m suspends$_j$)
from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound" neq-i-j ts_{sb}\cdot i
\cdot ts_{sb\cdot j}]
  have outstanding-refs is-Write_{sb} \cap outstanding-refs is-non-volatile-Read_{sb}
  suspends_j = {}
    by (simp add: suspends_j)
from reads-consistent-flush-independent [OF this reads-consis-m]
  have reads-consis-flush-m: reads-consistent True (acquired True ?take-sb j O_j)
    ?m-A suspends_j,
  hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb j O_j) ?m-A ys
    by (simp add: split-suspends_j reads-consistent-append)

from valid-history [OF j-bound" ts_{sb\cdot j}]
  have h-consis:
    history-consistent ?sbj (hd-prog p_j (?take-sb_j @?drop-sb_j)) (?take-sb_j @suspends_j)
    apply (simp only: split-suspends_j [symmetric] suspends_j)
  apply simp
  done

  have last-prog-hd-prog: last-prog (hd-prog p_j sb_j) ?take-sb_j = (hd-prog p_j suspends_j)
  proof -
    from last-prog have last-prog p_j (?take-sb_j @?drop-sb_j) = p_j
      by simp
  from last-prog-hd-prog-append'[OF h-consis] this
    have last-prog (hd-prog p_j suspends_j) ?take-sb_j = hd-prog p_j suspends_j
      by (simp only: split-suspends_j [symmetric] suspends_j)
  moreover
    have last-prog (hd-prog p_j (?take-sb_j @ suspends_j)) ?take-sb_j =
      last-prog (hd-prog p_j suspends_j) ?take-sb_j
        apply (simp only: split-suspends_j [symmetric] suspends_j)
        by (rule last-prog-hd-prog-append)
  ultimately show ?thesis
    by (simp add: split-suspends_j [symmetric] suspends_j)
  qed

from valid-write-sops [OF j-bound" ts_{sb\cdot j}]
  have \forall sop\in write-sops (?take-sb_j @?suspends). valid-sop sop
    by (simp add: split-suspends_j [symmetric] suspends_j)
  then obtain valid-sops-take: \forall sop\in write-sops ?take-sb_j. valid-sop sop and
    valid-sops-drop: \forall sop\in write-sops ys. valid-sop sop
      apply (simp only: write-sops-append )
      apply auto
      done

from read-tmps-distinct [OF j-bound" ts_{sb\cdot j}]
  have distinct-read-tmps (?take-sb_j @suspends_j)
    by (simp add: split-suspends_j [symmetric] suspends_j)
  then obtain
    read-tmps-take-drop: read-tmps ?take-sb_j \cap read-tmps suspends_j = {}
    and
distinct-read-tmps-drop: distinct-read-tmps suspends

apply (simp only: split-suspends \[ symmetric \] suspends)
apply (simp only: distinct-read-tmps-append)
done

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog
have hist-consis: history-consistent \( \theta_{sbj} \) (hd prog \( p_j \) suspends\( j \)) suspends\( j \)
by (simp add: split-suspends \[ symmetric \] suspends)
from reads-consistent-drop-volatile-writes-no-volatile-reads [OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read \( sbys \) = \{ \}
by (auto simp add: outstanding-refs-append suspends \[ symmetric \] split-suspends)
from flush-store-buffer-append [OF j-bound"" is\( j \) [simplified split-suspends] cph [simplified suspends]
hist-consis' [simplified split-suspends] valid-sops-drop distinct-read-tmps-drop [simplified split-suspends]
no-volatile-Read\( _{ab} \)-volatile-reads-consistent [OF no-vol-read], where \( S = ?share-A \)
obtain is\( j' \) \( R_j' /\ where \)
is\( j' \): instrs (Write\( _{ab} \) True a' sop v A' L' R' W' # zs) @ is\( sbj \) =
is\( j' \) @ prog-instrs (Write\( _{ab} \) True a' sop v A' L' R' W' # zs) and
steps-ys: (?ts-A, ?m-A, ?share-A) \( \Rightarrow_d^* \)
(?ts-A\[j:= (last-prog (hd-prog p_j (Write\( _{ab} \) True a' sop v A' L' R' W' # zs))) ys, 
is\( j' \),
\( \theta_{sbj} \) \( \mid \) (dom \( \theta_{sbj} \) - read-tmps (Write\( _{ab} \) True a' sop v A' L' R' W' # zs)),(),
\( D_j \lor \) outstanding-refs is-volatile-Write\( _{ab} \) ys \( \neq \) \{ \}, acquired True ys (acquired True ?take-sb\( j ) O_j, R_j' \)
flush ys ?m-A, share ys ?share-A)
(is (\( \vdash \), \( \vdash \) \( \vdash \) \( \vdash \)) \( \Rightarrow_d^* \) (?ts-ys,?m-ys,?shared-ys))
by (auto)

note conflict-computation = rtranclp-trans [OF rtranclp-r-rtranclp [OF steps-flush-sb, OF store-step] steps-ys]
from cph
have causal-program-history is\( sbj \) ((ys @ [Write\( _{ab} \) True a' sop v A' L' R' W']) @ zs)
by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history is\( sbj \) zs.
interpret causal\': causal-program-history is\( sbj \) zs by (rule cph')

from causal\'.causal-program-history [of []], simplified, OF refl] is\( j' \)
obtain is\( j'' \)
where is\( j'' \) = Write True a' sop A' L' R' W' #is\( j'' \) and
is\( j'' \): instrs zs @ is\( sbj \) = is\( j'' \) @ prog-instrs zs

598
by clarsimp

from j-bound''
have j-bound-ys: j < length ?ts-ys
  by auto

from j-bound-ys neq-i-j
have ts-ys-j: ?ts-ys?j=(last-prog (hd-prog p_j (Write sb True a' sop v A' L' R' W'# zs))
y, is_j',
  \( \partial_{sbj} \mid (\text{dom } \partial_{sbj} - \text{read-tmps (Write sb True a' sop v A' L' R' W'# zs)}),() \),
  \( \mathcal{D}_j \lor \text{outstanding-refs is-volatile-Write sb ys } \neq \{ \}, \),
  acquired True ys (acquired True ?take-sb \( \mathcal{O}_j \)),\( \mathcal{R}_j \))
  by auto

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified is_j]
have a-unowned:
\( \forall i \ < \ \text{length } ?ts-ys. \ j \neq i \rightarrow (\text{let } (\mathcal{O}_i) = \text{map owned } ?ts-ys?i \ \text{in } a' \notin \mathcal{O}_i) \)
  apply cases
  apply (auto simp add: Let-def is_{sb})
  done
from a' in a-unowned [rule-format, of i] neq-i-j i-bound' A-R
show False
  by (auto simp add: Let-def)
qed

thus ?thesis
by (auto simp add: Let-def)
qed

have A-unaquired-by-others:
\( \forall j < \text{length } (\text{map } \mathcal{O}-sb \ ts_{sb}). \ i \neq j \rightarrow \)
  (let (\( \mathcal{O}_j \), sb_j) = \text{map } \mathcal{O}-sb \ ts_{sb}! j
  in A \cap \text{all-acquired sb}_j = \{\})
  proof
  
  \{ 
  fix j \( \mathcal{O}_j \) sb_j
  assume j-bound: j < length (map owned ts_{sb})
  assume neq-i-j: i\neq j
  assume ts_{sb}-j: (map \( \mathcal{O}-sb \ ts_{sb}\)! j = (\( \mathcal{O}_j \),sb_j)
  assume conflict: A \cap \text{all-acquired sb}_j \neq \{\}
  have False
  proof
  from j-bound leq
  have j-bound': j < length (map owned ts)
    by auto
  from j-bound have j-bound''; j < length ts_{sb}
    by auto

599
from j-bound' have j-bound'": j < length ts by simp

from conflict obtain a' where
  a'\text{-}in: a' \in A \text{ and }
  a'\text{-}in-j: a' \in \text{all-acquired sb}_j
by auto

let ?take-sb_j = (takeWhile (Not \circ \text{is-volatile-Write} sb) sb_j)
let ?drop-sb_j = (dropWhile (Not \circ \text{is-volatile-Write} sb) sb_j)

from ts-sim [rule-format, OF j-bound'" ts_{sb,j} j-bound'"
  obtain p_j suspends_j = (p_j, \text{is}_{sb,j}, \theta_{sb,j}, sb_j, D_{sb,j}, O_j, R_j) \text{ and }
  suspends_j: suspends_j = ?drop-sb_j
  D_j: D_{sb,j} = (D_j \lor \text{outstanding-refs is-volatile-Write}_{sb} sb_j \neq \{\}) \text{ and }
  instrs \text{suspends_j @ is}_{sb,j} = \text{is}_j \lor \text{prog-instrs suspends_j and }
  ts_j: tsl_j = (\text{hd-prog} p_j \text{suspends}_j, \text{is}_j, \text{ds}_{sb,j} = \text{dom} \text{ds}_{sb,j} \setminus \text{read-temps suspends}_j,(),
  D_j, \text{acquired True ?take-sb_j O}_j, \text{release ?take-sb_j (dom S_{sb}) R}_j)
  apply \text{(cases ts_{sb,j})}
  apply \text{(force simp add: Let-def)}
done

from a'\text{-}in-j all-acquired-append [of ?take-sb_j ?drop-sb_j]
have a' \in \text{all-acquired ?take-sb_j \lor a' \in \text{all-acquired suspends}_j}
  by (auto simp add: suspends_j)
thus False
proof
  assume a' \in \text{all-acquired ?take-sb_j}
  with A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb,j} j-bound a'\text{-}in
  show False
by (auto dest: all-acquired-unshared-acquired)
next
  assume conflict-drop: a' \in \text{all-acquired suspends}_j
  from split-all-acquired-in [OF conflict-drop]
  show False
proof
  assume \exists sop a'' v ys zs A L R W.
  suspends_j = ys @ Write_{sb} True a'' sop v A L R W\# zs \land a' \in A
  then
  obtain a'' sop' v' ys zs A'L'R' W' where
  split-suspends_j: suspends_j = ys @ Write_{sb} True a'' sop' v' A'L'R' W'\# zs
  (is suspends_j = ?suspends) \text{ and }
  a'\text{-}A': a' \in A'
by auto

  from direct-memop-step.Ghost [\text{where } \vartheta = \vartheta_{sb} \text{ and } m=\text{flush ?drop-sb m}]
  have (Ghost A L R W\# is_{sb}',
\( \theta \sb \), flush ?\( \text{drop-sb} \) \( m \), \( D \sb \),
acquired True \( \text{sb} \) \( O \sb \), release \( \text{sb} \) (dom \( S \sb \) \( R \sb \), share ?\( \text{drop-sb} \) \( S \)) \( \rightarrow \)
(is \( \theta \sb \)′, \( \theta \sb \)′), flush ?\( \text{drop-sb} \) \( m \), \( D \sb \),
acquired True \( \text{sb} \) \( O \sb \) \( \cup \) \( A \) \( \rightarrow \) \( R \),
augment-rels (dom (share ?\( \text{drop-sb} \) \( S \sb \))) \( R \) (release \( \text{sb} \) (dom \( S \sb \) \( R \sb \)), share ?\( \text{drop-sb} \) \( S \)) \( \rightarrow \) (is \( \theta \sb \)′, \( \theta \sb \)′), flush ?\( \text{drop-sb} \) \( m \), \( D \sb \),
acquired True \( \text{sb} \) \( O \sb \) \( \cup \) \( A \) \( \rightarrow \) \( R \),
augment-rels (dom (share ?\( \text{drop-sb} \) \( S \sb \))) \( R \) (release \( \text{sb} \) (dom \( S \sb \) \( R \sb \)), share ?\( \text{drop-sb} \) \( S \)) \( \rightarrow \) (is \( \theta \sb \)′, \( \theta \sb \)′), flush ?\( \text{drop-sb} \) \( m \), \( D \sb \),
acquired True \( \text{sb} \) \( O \sb \) \( \cup \) \( A \) \( \rightarrow \) \( R \),
augment-rels (dom (share ?\( \text{drop-sb} \) \( S \sb \))) \( R \) (release \( \text{sb} \) (dom \( S \sb \) \( R \sb \)), share ?\( \text{drop-sb} \) \( S \)) \( \rightarrow \) (is \( \theta \sb \)′, \( \theta \sb \)′), flush ?\( \text{drop-sb} \) \( m \), \( D \sb \),
acquired True \( \text{sb} \) \( O \sb \) \( \cup \) \( A \) \( \rightarrow \) \( R \),
}\( \text{from} \) direct-computation.concurrent-step.Memop \{OF i-bound-ts’ \{simplified \( is \sb \) \} ts’ \{i \} \{simplified \( is \sb \) \} \} \{this \{simplified \( is \sb \) \} \}\} \{i-bound \} \{i-bound’ \}
i < length \( ?\text{ts-A} \)
\\{by \} \{simp \}\{add: \( is \sb \) \}\{\}
\\{from \} \{valid-program-history \{OF \} j-bound” \} \\{ts \sb \} \{j \} \{j-bound \} \{j-bound’ \}
\{i-bound \} \{i-bound’ \} \{i-bound’’ \} \{j-bound \} \{j-bound’’ \} \{j < length \?ts-A \}
\{by \} \{simp \}\{\}
\\{from \} \{valid-program-history \{OF \} j-bound” \} \\{ts \sb \} \{j \} \{j-bound \} \{j-bound’’ \} \{j < length \?ts-A \}
\{by \} \{simp \}\{\}
\\{from \} \{valid-program-history \{OF \} j-bound” \} \\{ts \sb \} \{j \} \{j-bound \} \{j-bound’’ \} \{i-bound \} \{i-bound’ \} \{i-bound’’ \} \{j-bound \} \{j-bound’’ \} \{j < length \?ts-A \}
\{by \} \{simp \}\{\}
\\{from \} \{valid-program-history \{OF \} j-bound” \} \\{ts \sb \} \{j \} \{j-bound \} \{j-bound’’ \} \{j < length \?ts-A \}
\{by \} \{simp \}\{\}
from valid-reads [OF j-bound'' ts_{sb-j}]

have reads-consis: reads-consistent False \( M_{sb} \) \( sb \).

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing \( S_{sb} \) j-bound'' \( ts_{sb-j} \)]

have reads-consis-m: reads-consistent True (acquired True ?take-sb \( O \)) \( M_\) \( suspends \)
by (simp add: \( M \) \( suspends \))

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound'' \( ts_{sb-i} \) ts_{sb-j}]

have outstanding-refs is-Write \( M_\) \( drop-sb \) \( \cap \) outstanding-refs is-non-volatile-Read \( M_\)

suspends \( j \) = \{\}
by (simp add: \( M \) \( suspends \))
from reads-consistent-flush-independent [OF this reads-consis-m]

have reads-consis-flush-m: reads-consistent True (acquired True ?take-sb \( O \)) ?\( m-A \) suspends \( j \).

hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb \( O \)) ?\( m-A \) ys
by (simp add: split-suspends \( j \) reads-consistent-append)
from valid-history [OF j-bound'' ts_{sb-j}]

have h-consis:

history-consistent \( \emptyset \) \( sbj \) (hd-prog \( p_j \) (?take-sb \( j \) @?drop-sb \( j \))) (?take-sb \( j \) @suspends \( j \))

apply (simp only: split-suspends \( j \) [symmetric] \( \) suspends \( j \))
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog \( p_j \) \( sb_j \)) ?take-sb \( j \) = (hd-prog \( p_j \) suspends \( j \))

proof 

from last-prog have last-prog \( p_j \) (?take-sb \( j \) @?drop-sb \( j \)) = \( p_j \)
by simp
from last-prog-hd-prog-append' [OF h-consis] this

have last-prog (hd-prog \( p_j \) suspends \( j \)) ?take-sb \( j \) = hd-prog \( p_j \) suspends \( j \)

by (simp only: split-suspends \( j \) [symmetric] \( \) suspends \( j \))
moreover

have last-prog (hd-prog \( p_j \) (?take-sb \( j \) @ suspends \( j \))) ?take-sb \( j \) =

last-prog (hd-prog \( p_j \) suspends \( j \)) ?take-sb \( j \)
apply (simp only: split-suspends \( j \) [symmetric] \( \) suspends \( j \))
by (rule last-prog-hd-prog-append)
ultimately show ?thesis

by (simp add: split-suspends \( j \) [symmetric] \( \) suspends \( j \))

qed

from valid-write-sops [OF j-bound'' ts_{sb-j}]

have \( \forall \) sop \( \in \) write-sops (?take-sb \( j \) @?suspends), valid-sop sop
by (simp add: split-suspends \( j \) [symmetric] \( \) suspends \( j \))
then obtain valid-sops-take: \( \forall \) sop\( \in \) write-sops ?take-sb\_j, valid-sop sop and valid-sops-drop: \( \forall \) sop\( \in \) write-sops ys, valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound’’ ts\_sb-j]
have distinct-read-tmps (?take-sb\_j@suspends\_j)
by (simp add: split-suspends\_j [symmetric] suspends\_j)
then obtain
read-tmps-take-drop: read-tmps ?take-sb\_j \cap read-tmps suspends\_j = \{\} and distinct-read-tmps-drop: distinct-read-tmps suspends\_j
apply (simp only: split-suspends\_j [symmetric] suspends\_j)
apply (simp only: distinct-read-tmps-append)
done

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]

last-prog-hd-prog
have hist-consis’: history-consistent \( \hat{\delta}_{sbj} \) (hd-prog p\_j suspends\_j) suspends\_j
by (simp add: split-suspends\_j [symmetric] suspends\_j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read\_sb ys = \{\}
by (auto simp add: outstanding-refs-append suspends\_j [symmetric]
split-suspends\_j)

from flush-store-buffer-append [
OF j-bound’’’ is\_j [simplified split-suspends\_j] cph [simplified split-suspends\_j]
hist-consis’ [simplified split-suspends\_j] valid-sops-drop distinct-read-tmps-drop
[simplified split-suspends\_j]
no-volatile-Read\_sb-volatile-reads-consistent [OF no-vol-read], where
\( S = \{\_share-A\] 
obtain is\_j’ R\_j’ where
is\_j’: instrs (Write\_sb True a’’ sop’ v’ A’ L’ R’ W’ # zs) @ is\_sbj =
is\_j’ @ prog-instrs (Write\_sb True a’’ sop’ v’ A’ L’ R’ W’ # zs) and
steps-ys: (?ts-A, ?m-A, ?share-A) \( \Rightarrow \alpha^* \)
(?ts-A\_j): = (last-prog (hd-prog p\_j (Write\_sb True a’’ sop’ v’ A’ L’ R’ W’ # zs)) ys,
is\_j’,
\( \hat{\delta}_{sbj} \setminus \) (dom \( \hat{\delta}_{sbj} \) - read-tmps (Write\_sb True a’’ sop’ v’ A’ L’ R’ W’ # zs))).(),
\( D_j \lor \) outstanding-refs is-volatile-Write\_sb ys \( \neq \{\} \), acquired True ys
(acquired True ?take-sb\_j O\_j’T, R\_j’T’)],
flush ys ?m-A_share ys ?share-A)
(is (\cdot,\cdot,\cdot) \( \Rightarrow \alpha^* \) (?ts-ys,?m-ys,?shared-ys))
by (auto)

note conflict-computation = rtranclp-trans [OF rtranclp-r-rtranclp [OF steps-flush-sb, OF store-step] steps-ys]
from cph
have causal-program-history is_{sbj} ((ys @ [Write_{sb} True a'' sop' v' A' L' R' W']) @ zs)
by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history is_{sbj} zs.
interpret causal': causal-program-history is_{sbj} zs by (rule cph')
from causal'.causal-program-history [of [], simplified, OF refl] is_j'
obtain is_j''
where is_j': is_j' = Write True a'' sop' A' L' R' W'#is_j'' and
is_j'': instrs zs @ is_{sbj} = is_j'' @ prog-instrs zs
by clarsimp
from j-bound''
have j-bound-ys: j < length ?ts-ys
by auto
from j-bound-ys neq-i-j
have ts-ys-j: ?ts-ys![j]=(last-prog (hd-prog p_j (Write_{sb} True a'' sop' v' A' L' R' W'# zs)) ys, is_j',
\vartheta_{sbj} \mid (dom \vartheta_{sbj} - read-tmps (Write_{sb} True a'' sop' v' A' L' R' W'# zs))).(),
\mathcal{D}_j \lor outstanding-refs is-volatile-Write_{sb} ys \neq \{\},
acquired True ys (acquired True ?take-sb_j O_j),R_j)
by auto
from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).
from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified is_j']
have A'-unowned:
\forall i < length ?ts-ys. j\neq i \longrightarrow (let (O_i) = map owned ?ts-ys!i in A' \cap O_i = \{\})
apply cases
apply (fastforce simp add: Let-def is_{sb})+
done
from a'-in a'-A' A'-unowned [rule-format, of i] neq-i-j i-bound' A-R
show False
by (auto simp add: Let-def)
next
assume \exists A L R W ys zs.
suspends_j = ys @ Ghost_{sb} A L R W # zs \land a' \in A
then
obtain ys zs A' L' R' W' where
split-suspends_j: suspends_j = ys @ Ghost_{sb} A' L' R' W'# zs (is suspends_j = ?suspends)
and a'-A': a' \in A'
by auto
from direct-memop-step.Ghost [where \vartheta=\vartheta_{sb} and m=flush ?drop-sb m]
have (Ghost A L R W# is_{sb}'),
\[ \varnothing_{sb}, (\cdot), \text{flush } ?\text{drop-sb } m, D_{sb}, \]
acquired True sb \( O_{sb} \), release sb (dom \( S_{sb} \)) \( R_{sb} \), share ?\text{drop-sb } S \rightarrow \\
(is_{sb}', \varnothing_{sb}, (\cdot), \text{flush } ?\text{drop-sb } m, D_{sb}, \\
acquired True sb \( O_{sb} \cup A \rightarrow R, \\
\text{augment-rels (dom (share ?\text{drop-sb } S)) } R \text{ (release sb (dom } S_{sb} \text{) } R_{sb} \text{), } \\
\text{share ?\text{drop-sb } } S \oplus W \ominus A \ominus L). \]

\text{from direct-computation.concurrent-step.Memop [OF } \\
i-bound-ts'[\text{simplified is}_{sb}] \text{ ts}^{\prime}i[\text{simplified is}_{sb}] \text{ this [simplified is}_{sb}] \\
\text{have store-step: } (?ts^{\prime}, \text{flush } ?\text{drop-sb } m, \text{share } ?\text{drop-sb } S) \Rightarrow_{d} \\
(?ts'[i := (p_{sb}, is_{sb}', \varnothing_{sb}, (\cdot), D_{sb}), \text{acquired True sb } O_{sb} \cup A \rightarrow R, \text{augment-rels} \\
\text{dom (share ?\text{drop-sb } S)) } R \text{ (release sb (dom } S_{sb} \text{) } R_{sb} \text{), } \\
\text{flush } ?\text{drop-sb } m, \text{share } ?\text{drop-sb } S \oplus W \ominus A \ominus L) \\
(is - \Rightarrow_{d} (?ts-A, ?m-A, \text{?share-A})) \\
\text{by (simp add: is}_{sb}] \\
\]

\text{from i-bound' have i-bound''; i < length } ?ts-A \\
\text{by simp} \\
\]

\text{from valid-program-history [OF j-bound’’ ts}_{sb\cdot j}] \\
\text{have causal-program-history is}_{sb\cdot j} \text{ sb}_{j}, \\
\text{then have cph: causal-program-history is}_{sb\cdot j} \text{ ?suspends} \\
\text{apply –} \\
\text{apply (rule causal-program-history-suffix [where sb=?take-sb}_{j} ) } \\
\text{apply (simp only: split-suspends}_{j}[\text{symmetric} \text{ suspends}_{j}] \\
\text{apply (simp add: split-suspends}_{j} \\
\text{done} \\
\]

\text{from ts}_{j} \text{ neq-i-j j-bound} \\
\text{have ts-A}_{j}:: ?ts-A[1] = (hd-prog p_{j} (ys @ Ghost_{sb} A' L' R' W'\# zs), is_{j}, \\
\varnothing_{sbj} | (dom } \varnothing_{sbj} - \text{read-tmps (ys @ Ghost}_{sb} A' L' R' W'\# zs)), (\cdot), D_{j}, \\
\text{acquired True ?take-sb}_{j} O_{j}, \text{release } ?\text{take-sb}_{j} (\text{dom } S_{sb}) \text{ R}_{j}) \\
\text{by (simp add: split-suspends}_{j}) \\
\]

\text{from j-bound’’’ i-bound’ neq-i-j have j-bound’’’; j < length } ?ts-A \\
\text{by simp} \\
\]

\text{from valid-last-prog [OF j-bound’’ ts}_{sb\cdot j}] \text{ have last-prog: last-prog p}_{j} \text{ sb}_{j} = p_{j}, \\
\text{then} \\
\text{have lp: last-prog p}_{j} \text{ ?suspends } = p_{j} \\
\text{apply –} \\
\text{apply (rule last-prog-same-append [where sb=?take-sb}_{j}] \\
\text{apply (simp only: split-suspends}_{j}[\text{symmetric} \text{ suspends}_{j}] \\
\text{apply simp} \\
\text{done} \\
\text{from valid-reads [OF j-bound’’ ts}_{sb\cdot j}] \\
\text{have reads-consis: reads-consistent False } O_{j} m_{sbj} sb_{j}, \\
\]

605


from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing
\(S_{sb}\) ts\(_{sb}\) j-bound"
] ts\(_{sb}\)-j reads-consis

have reads-consis-m: reads-consistent True (acquired True \(?\text{take-sb}_j \ O_j\) \(m\) suspends\(_j\)
by (simp add: \(m\) suspends\(_j\))

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound neq-i-j ts\(_{sb}\)-i ts\(_{sb}\)-j]
have outstanding-refs is-Write\(_{sb}\) ?drop-sb \(\cap\) outstanding-refs is-non-volatile-Read\(_{sb}\)
suspends\(_j\) = {}
by (simp add: suspends\(_j\))
from reads-consistent-flush-independent [OF this reads-consis-m]
have reads-consis-flush-m: reads-consistent True (acquired True \(?\text{take-sb}_j \ O_j\) \(?m-A\) suspends\(_j\),

hence reads-consis-m-A-ys: reads-consistent True (acquired True \(?\text{take-sb}_j \ O_j\) \(?m-A\) ys

by (simp add: split-suspends\(_j\) reads-consistent-append)

from valid-history [OF j-bound" ts\(_{sb}\)-j]
have h-consis:
history-consistent \(q_{sbj}\) (hd-prog \(p_j\) \(?\text{take-sb}_j @\text{suspends}_j\)) \(?\text{take-sb}_j @\text{suspends}_j\)
apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
apply simp

done

have last-prog-hd-prog: last-prog (hd-prog \(p_j\) sb\(_j\)) \(?\text{take-sb}_j\) = (hd-prog \(p_j\) suspends\(_j\))

proof –
from last-prog have last-prog \(p_j\) \(?\text{take-sb}_j @?\text{drop-sb}_j\) = \(p_j\)
by simp
from last-prog-hd-prog-append’ [OF h-consis] this
have last-prog (hd-prog \(p_j\) suspends\(_j\)) \(?\text{take-sb}_j\) = hd-prog \(p_j\) suspends\(_j\)
by (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
moreover
have last-prog (hd-prog \(p_j\) (?\text{take-sb}_j @ suspends\(_j\))) \(?\text{take-sb}_j\) =
last-prog (hd-prog \(p_j\) suspends\(_j\)) \(?\text{take-sb}_j\)
apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))
by (rule last-prog-hd-prog-append)
ultimately show \(?\)thesis
by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))

qed

from valid-write-sops [OF j-bound" ts\(_{sb}\)-j]
have \(\forall\) sop\(\in\) write-sops \(?\text{take-sb}_j @?\text{suspends}\), valid-sop sop
by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
then obtain valid-sops-take: \(\forall\) sop\(\in\) write-sops \(?\text{take-sb}_j\), valid-sop sop and
valid-sops-drop: \(\forall\) sop\(\in\) write-sops ys. valid-sop sop
apply (simp only: write-sops-append )
apply auto
done

606
\[\text{from read-tmps-distinct [OF } j\text{-bound'' } ts_{sbj}\text{-j}]\]
\[\text{have distinct-read-tmps (?take-sb}_{j}\text{-suspends}_{j})\]
\[\text{by (simp add: split-suspends}_{j} \text{ [symmetric] suspends}_{j})\]
\[\text{then obtain}\]
\[\text{read-tmps\text{-take\text{-drop}: read-tmps } ?\text{take-sb}_{j} \cap \text{read-tmps suspends}_{j} = \{\}\ and}\]
\[\text{distinct-read-tmps\text{-drop: distinct-read-tmps suspends}_{j}}\]
\[\text{apply (simp only: split-suspends}_{j} \text{ [symmetric] suspends}_{j})\]
\[\text{apply (simp only: distinct-read-tmps-append)}\]
\[\text{done}\]

\[\text{from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis}\]
\[\text{last-prog-hd-prog}\]
\[\text{have hist-consis': history-consistent } \varnothing_{sbj} \text{ (hd-prog } p_{j} \text{ suspends}_{j}\text{) suspends}_{j}\]
\[\text{by (simp add: split-suspends}_{j} \text{ [symmetric] suspends}_{j})\]
\[\text{from reads-consistent-drop-volatile-writes-no-volatile-reads [OF reads-consis]}\]
\[\text{have no-vol-read: outstanding-refs is-volatile-Read}_{sbys} = \{\}\]
\[\text{by (auto simp add: outstanding-refs-append suspends}_{j} \text{ [symmetric] split-suspends}_{j})\]
\[\text{from flush-store-buffer-append [ OF j-bound'' } ts_{sbj}\text{-j}]\]
\[\text{cph [simplified split-suspends}_{j}]\]
\[\text{ts-A-j [simplified split-suspends}_{j}]\text{ refl lp [simplified split-suspends}_{j}]\text{ reads-consis-m-A-ys}\]
\[\text{hist-consis': [simplified split-suspends}_{j}]\text{ valid-sops-drop distinct-read-tmps-drop}\]
\[\text{no-volatile-Read}_{sbj}\text{-volatile-reads-consistent [OF no-vol-read], where}\]
\[S=?\text{share-A}\]
\[\text{obtain } isj'_{s} = (isj'_{s} \Rightarrow d^* (ts_{ys}, m_{ys}, share_{ys} = \{\}))\]
\[\text{by (auto)}\]
\[\text{note conflict-computation } = rtranclp-trans [OF rtranclp-r-rtranclp [OF steps-flush-sb, OF store-step] steps-ys]\]
\[\text{from cph}\]
\[\text{have causal-program-history is}_{sbj} ((ys @ [Ghost}_{sbj} A' L' R' W'# zs) @ zs)\]
\[\text{by simp}\]
\[\text{from causal-program-history-suffix [OF this]}\]
\[\text{have cph': causal-program-history is}_{sbj} zs.}\]
\[\text{interpret causal}_{sbj}: causal-program-history is}_{sbj} zs \text{ by (rule cph')}\]

607
from causal\textsubscript{1}, causal-prog-history [of [], simplified, OF refl] is\textsubscript{j} \\
obtain is\textsubscript{j}'' \\
where is\textsubscript{j}'; is\textsubscript{j}'' = Ghost A' L' R' W'\#is\textsubscript{j}'' and \\
is\textsubscript{j}''': instrs zs @ is\textsubscript{sbj} = is\textsubscript{j}'' @ prog-instrs zs \\
by clarsimp \\

from j-bound'''' \\
have j-bound-ys: j < length ?ts-ys \\
by auto \\

from j-bound-ys neq-i-j \\
have ts-ys-j: ?ts-ys\textsubscript{j} = (last-prog (hd-prog p\textsubscript{j} (Ghost\textsubscript{sb} A' L' R' W'\# zs)) ys, is\textsubscript{j}'', \\
\theta_{\textsubscript{sbj}}' | \ (dom \theta_{\textsubscript{sbj}} - read-tmps (Write\textsubscript{sb} True a'' sop' v' A' L' R' W'\# zs)),(), \\
\mathcal{D}_j \lor outstanding-refs is-volatile-Write\textsubscript{sb} ys \neq \{\}, \\
acquired True ys (acquired True ?take-sb \textsubscript{j} \textit{O} \textsubscript{j}, \textit{R}_j') \\
by auto \\

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation] \\
have safe-delayed (?ts-ys, ?m-ys, ?shared-ys). \\

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified is\textsubscript{j}'] \\
have A\textasciitilde\textsubscript{\textit{unowned}}: \\
\forall i < length ?ts-ys, j\neq i \rightarrow (let (\textit{O}_i) = map owned ?ts-ys!i in A \cap \textit{O}_i = \{\}) \\
apply cases \\
apply (fastforce simp add: Let-def is\textsubscript{sb})+ \\
done \\
from a\textasciitilde\textsubscript{i} in a\textasciitilde\ A\textasciitilde\ A\textasciitilde\\textsubscript{\textit{unowned}} [rule-format, of i] neq-i-j i-bound' A-R \\
show False \\
by (auto simp add: Let-def) \\
qed \\
qed \\
qed \\
} \\
thus ?thesis \\
by (auto simp add: Let-def) \\
qed \\

have A-no-read-only-reads-by-others: \\
\forall j < length (map \textit{O}\textsubscript{sb} ts\textsubscript{sb}). i \neq j \rightarrow \\
\quad (let (\textit{O}_j, \textit{sb}_j) = map \textit{O}\textsubscript{sb} ts\textsubscript{sb}! j \\
in A \cap read-only-reads (acquired True (takeWhile (Not o is-volatile-Write\textsubscript{sb} \textit{sb}_j) \\
\textit{O}_j)) \\
\quad \quad (dropWhile (Not o is-volatile-Write\textsubscript{sb} \textit{sb}_j) \textit{sb}_j) = \{\}) \\
proof - \\
\{ \\
\fix j \textit{O}_j \textit{sb}_j \\
\assume j-bound: j < length (map owned ts\textsubscript{sb}) \\
\assume neq-i-j: i \neq j \\
\assume ts\textsubscript{sb}-j: (map \textit{O}\textsubscript{sb} ts\textsubscript{sb})\textsubscript{j} = (\textit{O}_j, \textit{sb}_j) \\
\}
let ?take-sbj = (takeWhile (Not ◦ is-volatile-Write sb) sb)
let ?drop-sbj = (dropWhile (Not ◦ is-volatile-Write sb) sb)

assume conflict: A ∩ read-only-reads (acquired True ?take-sbj O) ?drop-sbj ≠ {}
have False
proof –
from j-bound leq
have j-bound': j < length (map owned ts)
  by auto
from j-bound have j-bound'': j < length ts
  by auto
from j-bound' have j-bound'''': j < length ts
  by simp

from conflict obtain a' where
  a'-in: a' ∈ A and
  a'-in-j: a' ∈ read-only-reads (acquired True ?take-sbj O) ?drop-sbj
  by auto

from ts-sim [rule-format, OF j-bound'''] ts_sbj-j j-bound''
obtain p_j suspends_j is_sbj D_sbj D_j R_j ⊥_sbj is_j where
  ts_sbj-j: ts_sbj ! j = (p_j, is_sbj, ⊥_sbj, sb_j, D_sbj, O_j, R_j) and
  suspends_j: suspends_j = ?drop-sbj and
  is_j: instrs suspends_j @ is_sbj = is_j @ prog-instrs suspends_j and
  D_j: D_sbj = (D_j ∨ outstanding-refs is-volatile-Write sb j ≠ {}) and
  ts_j: ts! j = (hd-prog p_j suspends_j, is_j,
    ⊥_sbj | (dom ⊥_sbj − read-tmps suspends_j),(), D_j, acquired True ?take-sbj O_j, release
    ?take-sbj (dom S_{sb}) R_j)
  apply (cases ts_sbj[j])
  apply (force simp add: Let-def)
done

from split-in-read-only-reads [OF a'-in-j [simplified suspends_j [symmetric]]]
obtain t v ys zs where
  split-suspends_j: suspends_j = ys @ Read_{sb} False a' t v# zs (is suspends_j = ?suspends)
  and
  a'-inacq: a' ∉ acquired True ys (acquired True ?take-sbj O)
  by blast

from direct-memop-step.Ghost [where ⊥=⊥_{sb} and m=flush ?drop-sb m]
have (Ghost A L R W# ⊥_{sb}′,
  ⊥_{sb}, (), flush ?drop-sb m, D_{sb},
  acquired True sb O_{sb}, release sb (dom S_{sb}) R_{sb}, share ?drop-sb S) →
  (is_{sb}, ⊥_{sb}, (), flush ?drop-sb m, D_{sb},
  acquired True sb O_{sb} ∪ A - R,
  augment-rels (dom (share ?drop-sb S)) R (release sb (dom S_{sb}) R_{sb}),
  share ?drop-sb S ⊎ W R ⊎ A L).

609
from direct-computation.concurrent-step.Memop [OF
i-bound-ts′ [simplified is\_sb] ts′\_i [simplified is\_sb] this [simplified is\_sb]]

have store-step: (?ts′, flush ?drop-sb m, share ?drop-sb S) \Rightarrow_d
         (?ts′\_i := (p_{sb}, is_{sb}′, 0_{sb}, ()}, D_{sb}, acquired True sb O_{sb} \cup A - R, augment-rels
         (dom (share ?drop-sb S)) R (release sb (dom S_{sb} R_{sb}))),
         flush ?drop-sb m, share ?drop-sb S \oplus W R \ominus A L)
(is - \Rightarrow_d (?ts-A, ?m-A, ?share-A))
by (simp add: is\_sb)

from i-bound′ have i-bound″: i < length ?ts-A
  by simp

from valid-program-history [OF j-bound″ ts_{sb-j}]
have causal-program-history is_{sbj} sb_j,
then have cph: causal-program-history is_{sbj} ?suspends
  apply –
  apply (rule causal-program-history-suffix [where sb=?take-sb_j] )
  apply (simp only: split-suspends_j [symmetric] suspends_j)
  apply (simp add: split-suspends_j)
done

from ts\_j neq-i-j j-bound
have ts-A-j: ?ts-A!j = (hd-prog p_j (ys @ Read sb False a′ t v# zs), is_j,
  \_sbj |′ (dom 0_{sbj} - read-tmps (ys @ Read sb False a′ t v# zs)), (), D_j,
  acquired True ?take-sb_j O_j, release ?take-sb_j (dom S_{sb} R_j))
by (simp add: split-suspends_j)

from j-bound″′ i-bound′ neq-i-j have j-bound″′: j < length ?ts-A
  by simp

from valid-last-prog [OF j-bound″ ts_{sb-j}]
have last-prog: last-prog p_j sb_j = p_j,
then have lp: last-prog p_j ?suspends = p_j
  apply –
  apply (rule last-prog-same-append [where sb=?take-sb_j])
  apply (simp only: split-suspends_j [symmetric] suspends_j)
  apply simp
done

from valid-reads [OF j-bound″ ts_{sb-j}]
have reads-consis: reads-consistent False O_j m_{sb} sb_j,

from reads-consistent-flush-all-until-volatile-write [OF \\ (valid-ownership-and-sharing
S_{sb} ts\_sb′ j-bound″
\ts_{sb-j} reads-consis]
have reads-consis-m: reads-consistent True (acquired True ?take-sb_j O_j) m suspends_j
by (simp add: m suspends_j)
from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound" neq-i-j ts_{sb-i} ts_{sb-j}]
  have outstanding-refs is-Write_{sb} ?drop-sb ∩ outstanding-refs is-non-volatile-Read_{sb} suspends_{sb j} = {}
    by (simp add: suspends_{sb j})
from reads-consistent-flush-independent [OF this reads-consis-m]
  have reads-consis-flush-m: reads-consistent True (acquired True ?take-sb_{j} O_{j}) ?m-A suspends_{j}.
  hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb_{j} O_{j}) ?m-A ys
    by (simp add: split-suspends_{j} reads-consistent-append)

from valid-history [OF j-bound" ts_{sb-j}]
  have h-consis:
    history-consistent θ_{sbj} (hd-prog p_{j} (?take-sb_{j} @suspends_{j})) (?take-sb_{j} @suspends_{j})
    apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
    apply simp
    done

  have last-prog-hd-prog: last-prog (hd-prog p_{j} sb_{j} ) ?take-sb_{j} = (hd-prog p_{j} suspends_{j})
    proof —
    from last-prog have last-prog p_{j} (?take-sb_{j}@?drop-sb_{j}) = p_{j}
      by simp
      from last-prog-hd-prog-append' [OF h-consis] this
      have last-prog (hd-prog p_{j} suspends_{j} ) ?take-sb_{j} = hd-prog p_{j} suspends_{j}
      by (simp only: split-suspends_{j} [symmetric] suspends_{j})
      moreover
      have last-prog (hd-prog p_{j} (?take-sb_{j} @ suspends_{j})) ?take-sb_{j} =
        last-prog (hd-prog p_{j} suspends_{j} ) ?take-sb_{j}
      apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
      by (rule last-prog-hd-prog-append)
      ultimately show ?thesis
      by (simp add: split-suspends_{j} [symmetric] suspends_{j})
      qed

  from valid-write-sops [OF j-bound" ts_{sb-j}]
  have ∀ sop∈write-sops (?take-sb_{j}@?suspends_{j}). valid-sop sop
    by (simp add: split-suspends_{j} [symmetric] suspends_{j})
  then obtain valid-sops-take: ∀ sop∈write-sops ?take-sb_{j}. valid-sop sop and
  valid-sops-drop: ∀ sop∈write-sops ys. valid-sop sop
    apply (simp only: write-sops-append )
    apply auto
    done

  from read-tmps-distinct [OF j-bound" ts_{sb-j}]
  have distinct-read-tmps (?take-sb_{j}@suspends_{j})
    by (simp add: split-suspends_{j} [symmetric] suspends_{j})
  then obtain
  read-tmps-take-drop: read-tmps ?take-sb_{j} ∩ read-tmps suspends_{j} = {} and

611
distinct-read-tmps-drop: distinct-read-tmps suspends_j

apply (simp only: split-suspends_j [symmetric] suspends_j)
apply (simp only: distinct-read-tmps-append)
done

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]
last-prog-hd-prog
have hist-consis': history-consistent θ sbj (hd-prog p_j suspends_j) suspends_j
  by (simp add: split-suspends_j [symmetric] suspends_j)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read sbj ys = {}
  by (auto simp add: outstanding-refs-append suspends_j [symmetric]
    split-suspends_j)

from flush-store-buffer-append [OF j-bound'''''''' i'_j [simplified split-suspends_j] cph [simplified suspends_j]
    hist-consis' [simplified split-suspends_j] valid-sops-drop distinct-read-tmps-drop
  [simplified split-suspends_j]
no-volatile-Read sbj-volatile-reads-consistent [OF no-vol-read], where
S=?share-A]
  obtain is''_j' R'j' where
    is''_j': instrs (Read sbj False a' t v # zs) @ is''_sbj =
    is''_j' @ prog-instrs (Read sbj False a' t v # zs) and
    steps-ys: (?ts-A, ?m-A, ?share-A) ⇒ d''
    (?ts-A[j]:= (last-prog (hd-prog p_j (Ghost sb A' L' R' W'# zs)) ys,
      is''_j',
      θ sbj |' (dom θ sbj − read-tmps (Read sbj False a' t v # zs)),().
      D_j ∨ outstanding-refs is-volatile-Write sbj ys ≠ {}, acquired True ys (acquired
      True ?take-sb o_j), R''_j') ],
    flush ys ?m-A,
    share ys ?share-A)
    (is (-,-,-) ⇒ d'' (?ts-ys, ?m-ys, ?shared-ys))
  by (auto)

note conflict-computation = rtranclp-trans [OF rtranclp-r-rtranclp [OF steps-flush-sb,
  OF store-step] steps-ys]

from cph
have causal-program-history is''_sbj ((ys @ [Read sbj False a' t v]) @ zs)
  by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history is''_sbj zs.
interpret causal_j: causal-program-history is''_sbj zs by (rule cph')

from causal_j.causal-program-history [of [], simplified, OF refl] is''_j'
obtain is''_j''
  where is''_j' = Read False a' t # is''_j'' and
  is''_j'': instrs zs @ is''_sbj = is''_j'' @ prog-instrs zs

612
by clarsimp

from j-bound''
have j-bound-ys: j < length ?ts-ys
  by auto

from j-bound-ys neq-i-j
have ts-ys-j: ?ts-ys!j=(last-prog (hd-prog p j (Read sb False a' t v# zs)) ys, isj',
  \theta_{sbj} \upharpoonright \dom \theta_{sbj} - \read-tmps (Read sb False a' t v# zs)),(),
  D_j \vee \outstanding-refs is-volatile-Write sb ys \neq \emptyset,
  acquired True ys (acquired True ?take-sb j O_j),(R_j')
  by auto

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified isj']
have a' \in acquired True ys (acquired True ?take-sb j O_j) \lor
  a' \in read-only (share ys (share ?drop-sb S \oplus W R \ominus A L))
  apply cases
  apply (auto simp add: Let-def is sb)
done
with a'-unacq
have a'-ro: a' \in read-only (share ys (share ?drop-sb S \oplus W R \ominus A L))
  by auto
from a'-in
have a'-not-ro: a' \notin read-only (share ?drop-sb S \oplus W R \ominus A L)
  by (auto simp add: in-read-only-convs)

have a' \in O_j \cup all-acquired sb
proof -
{ assume a-notin: a' \notin O_j \cup all-acquired sb
from weak-sharing-consis [OF j-bound'' ts sb-j]
have weak-sharing-consistent O_j sb_j.
with weak-sharing-consistent-append [of O_j ?take-sb j ?drop-sb j]
have weak-sharing-consistent (acquired True ?take-sb j O_j) suspends_j
  by (auto simp add: suspends_j)
with split-suspends_j
have weak-consis: weak-sharing-consistent (acquired True ?take-sb j O_j) ys
  by (simp add: weak-sharing-consistent-append)
from all-acquired-append [of ?take-sb j ?drop-sb j]
have all-acquired ys \subseteq all-acquired sb
  apply (clarsimp)
  apply (clarsimp simp add: suspends_j [symmetric] split-suspends_j all-acquired-append)
done
with a-notin acquired-takeWhile-non-volatile-Write sb [of sb_j O_j]
  all-acquired-append [of ?take-sb j ?drop-sb j]
have a' \notin acquired True (takeWhile (Not \circ is-volatile-Write sb) sb_j) O_j \cup all-acquired ys
  by auto

613
from read-only-share-unowned [OF weak-consis this a'-ro] 
have a' ∈ read-only (share ?drop-sb S ⊕ W ⊕ A L).

with a'-not-ro have False 
  by auto 
} 
  thus ?thesis by blast 
qed 

moreover 
from A-unquired-by-others [rule-format, OF - neq-i-j] ts_{sb,j} j-bound 
have A ∩ all-acquired sb_j = {} 
  by (auto simp add: Let-def) 
moreover 
from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb,j} j-bound 
have A ∩ O_j = {} 
  by (auto simp add: Let-def dest: all-shared-acquired-in) 
moreover note a'-in 
ultimately 
show False 
  by auto 
qed 

thus ?thesis 
  by (auto simp add: Let-def) 
qed 

have valid-own': valid-ownership S_{sb'} ts_{sb'} 
proof (intro-locales) 
show outstanding-non-volatile-refs-owned-or-read-only S_{sb'} ts_{sb'} 
proof 
  from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb,i}] 
  have non-volatile-owned-or-read-only False S_{sb} O_{sb} (sb @ [Ghost sb A L R W]) 
    by (auto simp add: non-volatile-owned-or-read-only-append) 
  from outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this] 
  show ?thesis 
    by (simp add: ts_{sb'} sb' O_{sb'} S_{sb'}) 
qed 

next 
show outstanding-volatile-writes-unowned-by-others ts_{sb'} 
proof (unfold-locales) 
  fix i_1 j p_1 is_1 O_1 R_1 D_1 xs_1 sb_1 p_j is_j O_j R_j D_j xs_j sb_j 
  assume i_1-bound: i_1 < length ts_{sb'} 
  assume j-bound: j < length ts_{sb'} 
  assume i_1\neq j 
  assume ts-i_1: ts_{sb} i_1 = (p_1,is_1,xs_1, sb_1, D_1, O_1, R_1) 
  assume ts-j: ts_{sb} j = (p_j,is_j,xs_j, sb_j, D_j, O_j, R_j) 
  show (O_j ∪ all-acquired sb_j) ∩ outstanding-refs is-volatile-Write_{sb} sb_1 = {} 
proof (cases i_1=i) 
  case True

614
with i_1-j have i-j: i ≠ j
  by simp

from j-bound have j-bound': j < length ts_{sb}
  by (simp add: ts_{sb}′)

hence j-bound'': j < length (map owned ts_{sb})
  by simp

from ts-j i-j have ts-j': ts_{sb}'[j] = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
  by (simp add: ts_{sb}′)

from outstanding-volatile-writes-unowned-by-others
  [OF i-bound j-bound' i-j ts_{sb}-i ts_{sb}′]
have (O_j ∪ all-acquired sb_j) ∩ outstanding-refs is-volatile-Write_{sb} sb = {}.

with ts-i_1 True i-bound show ?thesis
  by (clarsimp simp add: ts_{sb}' sb′ outstanding-refs-append acquired-takeWhile-non-volatile-Write_{sb})

next
  case False
  note i_1-i = this

  from i_1-bound have i_1-bound': i_1 < length ts_{sb}
    by (simp add: ts_{sb}′)

  from ts-i_1 False have ts-i_1': ts_{sb}''[i_1] = (p_{i_1}, is_{i_1}, xs_{i_1}, sb_{i_1}, D_{i_1}, O_{i_1}, R_{i_1})
    by (simp add: ts_{sb}′)

  show ?thesis
  proof (cases j=i)
    case True
    from i_1-bound' have i_1-bound'': i_1 < length (map owned ts_{sb})
      by simp

    from outstanding-volatile-writes-unowned-by-others
      [OF i_1-bound' i-bound-i ts-i_1′ ts_{sb}]
    have (O_{sb} ∪ all-acquired sb) ∩ outstanding-refs is-volatile-Write_{sb} sb_{i_1} = {}.

    moreover
    from A-unused-by-others [rule-format, OF - False [symmetric]] False ts-i_1 i_1-bound
    have A ∩ outstanding-refs is-volatile-Write_{sb} sb_{i_1} = {}
    by (auto simp add: Let-def ts_{sb}′)

    ultimately
    show ?thesis
    using ts-j True ts_{sb}′
    by (auto simp add: i-bound ts_{sb}' O_{sb}' sb′ all-acquired-append)

next
    case False
    from j-bound have j-bound': j < length ts_{sb}

    by (simp add: ts_{sb}′)

    from ts-j False have ts-j': ts_{sb}'[j] = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
    by (simp add: ts_{sb}′)

    from outstanding-volatile-writes-unowned-by-others
show \((O_j \cup \text{all-acquired } sb_j) \cap \text{outstanding-refs is-volatile-Write}_{sb} \sb_1 = \{\}\).

qed

next
show read-only-reads-unowned ts_{sb}′

proof
fix \(n \ m\)
fix \(p_n \ is_n \ O_n \ R_n \ D_n \ \vartheta_n \ sb_n \ p_m \ is_m \ O_m \ R_m \ D_m \ \vartheta_m \ sb_m\)

assume \(n\)-bound: \(n < \text{length } ts_{sb}'\)
and \(m\)-bound: \(m < \text{length } ts_{sb}'\)
and \(\text{neq-n-m: } n \neq m\)
and \(\text{nth: } ts_{sb}'!n = (p_n, is_n, \vartheta_n, sb_n, D_n, O_n, R_n)\)
and \(\text{mth: } ts_{sb}'!m = (p_m, is_m, \vartheta_m, sb_m, D_m, O_m, R_m)\)

from \(n\)-bound have \(n\)-bound′: \(n < \text{length } ts_{sb}'\) by (simp add: ts_{sb}′)
from \(m\)-bound have \(m\)-bound′: \(m < \text{length } ts_{sb}'\) by (simp add: ts_{sb}′)

show \((O_m \cup \text{all-acquired } sb_m) \cap \text{read-only-reads (acquired True (takeWhile (Not o is-volatile-Write}_{sb} sb_n) O_n) (dropWhile (Not o is-volatile-Write}_{sb} sb_n)}) = \{\}\)
proof (cases \(m=i\))
case True
with \(\text{neq-n-m: } n \neq m\)
by auto

with \(n\)-bound \(\text{nth i-bound: } ts_{sb}'!n = (p_n, is_n, \vartheta_n, sb_n, D_n, O_n, R_n)\)
by (auto simp add: ts_{sb}′)

note \(\text{read-only-reads-unowned } [\text{OF } n\text{-bound' i-bound neq-n-i nth' ts}_{sb}-i]\)

moreover
from \(\text{A-no-read-only-reads-by-others } [\text{rule-format, OF - neq-n-i } [\text{symmetric}]] n\text{-bound'} \text{nth'}\)

have \(\text{A } \cap \text{read-only-reads (acquired True (takeWhile (Not o is-volatile-Write}_{sb} sb_n) O_n) (dropWhile (Not o is-volatile-Write}_{sb} sb_n)}) = \{\}\)
by auto
ultimately
show \(?\)thesis
using True \(ts_{sb}-i\) \(\text{nth' mth n-bound' m-bound'}\)
by (auto simp add: ts_{sb}′ \(O_{sb}' \ sb'\text{-all-acquired-append}\))

next
case False

note \(\text{neq-m-i = } \text{this}\)

with \(m\)-bound \(\text{mth i-bound: } ts_{sb}'!m = (p_m, is_m, \vartheta_m, sb_m, D_m, O_m, R_m)\)
by (auto simp add: ts_{sb}′)

show \(?\)thesis
proof (cases \(\text{n=i}\))
case True

note \(\text{read-only-reads-unowned } [\text{OF } i\text{-bound m-bound' neq-m-i } [\text{symmetric}]} ts_{sb}-i\) \(\text{mth'}\)
then show ?thesis
using True neq-m-i ts\sb\^i nth m-bound’ m-bound’
apply (case-tac outstanding-refs (is-volatile-Write sb) sb = { })
apply (clarsimp simp add: outstanding-vol-write-take-drop-append
acquired-append read-only-reads-append ts\sb\^i sb’ O_{sb}’)
done

next

with n-bound nth i-bound have nth’: ts\sb\^n = (p_n, is_n, \emptyset_n, sb_n, D_n, O_n, R_n)
by (auto simp add: ts\sb\^i)
from read-only-reads-unowned [OF n-bound’ m-bound’ neq-n-m nth’ mth’] False neq-m-i

show ?thesis
by (clarsimp)
qed

next

show ownership-distinct ts\sb\^i
proof –

have \forall j<length ts\sb\^i. i \neq j \rightarrow
(let (p_j, is_j, \emptyset_j, sb_j, D_j, O_j, R_j) = ts\sb\^j ! j
in (O_{sb} \cup all-acquired sb’) \cap (O_j \cup all-acquired sb_j) = { })
proof –

{ 
  fix j p_j is_j \emptyset_j D_j sb_j
  assume neq-i-j: i \neq j
  assume j-bound: j < length ts\sb
  assume ts\sb\^j: ts\sb\^j ! j = (p_j, is_j, \emptyset_j, sb_j, D_j, O_j, R_j)
  have (O_{sb} \cup all-acquired sb’) \cap (O_j \cup all-acquired sb_j) = { }
  proof –

  { 
    fix a’
    assume a’-in-i: a’ \in (O_{sb} \cup all-acquired sb’)
    assume a’-in-j: a’ \in (O_j \cup all-acquired sb_j)
    have False
    proof –
      from a’-in-i have a’ \in (O_{sb} \cup all-acquired sb) \lor a’ \in A
      by (simp add: sb’ all-acquired-append)
    then show False
    proof
      assume a’ \in (O_{sb} \cup all-acquired sb)
      with ownership-distinct [OF i-bound j-bound neq-i-j ts\sb\^i ts\sb\^j] a’-in-j
      show ?thesis
      by auto
    next
      assume a’ \in A
      moreover
      have j-bound’: j < length (map owned ts\sb)
      using j-bound by auto
  }
from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb\cdot j} \ j-bound

obtain A \cap \text{acquired} True (\text{takeWhile \ NOT \ is-volatile-Write}_{sb\cdot j} O_1 = \{\}) \ \text{and}
A \cap \text{all-shared} \ (\text{takeWhile \ NOT \ is-volatile-Write}_{sb\cdot j} O_{sb\cdot j}) = \{\}

by \ (\text{auto \ simp \ add: \ Let-def})

moreover

from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb\cdot j} \ j-bound

have A \cap \text{all-acquired} sb_{j} = \{\}

by \ auto

ultimately

show \ ?thesis

using a'\cdot \text{-in-j}

by \ (\text{auto \ dest: \ all-shared-acquired-in})

qed

qed

from ownership-distinct-nth-update [OF i-bound ts_{sb\cdot i} this]

show \ ?thesis \ by \ (\text{simp \ add: \ ts_{sb\cdot j}' O_{sb\cdot j}' sb'})

qed

have \ valid-hist': \ valid-history \ program-step \ ts_{sb'}

proof
  from \ valid-history \ [OF \ i-bound \ ts_{sb\cdot i}]

  have \ history-consistent \ \emptyset_{sb} \ (\text{hd-prog} \ p_{sb} \ sb) \ sb.

  with \ valid-write-sops \ [OF \ i-bound \ ts_{sb\cdot i}]

  \text{valid-implies-valid-prog-hd} \ [OF \ i-bound \ ts_{sb\cdot i} \ valid]

  have \ history-consistent \ \emptyset_{sb} \ (\text{hd-prog} \ p_{sb} (sb\@[\text{Ghost}_{sb} A L R W]))

  \ (sb\@ [\text{Ghost}_{sb} A L R W])

  apply -

  apply \ (\text{rule \ history-consistent-appendI})

  apply \ (\text{auto \ simp \ add: \ hd-prog-append-Ghost_{sb}})

  done

from \ valid-history-nth-update \ [OF \ i-bound \ this]

show \ ?thesis \ by \ (\text{simp \ add: \ ts_{sb\cdot j}' sb' \ \emptyset_{sb'}})

qed

have \ valid-reads': \ valid-reads \ m_{sb} \ ts_{sb'}

proof
  from \ valid-reads \ [OF \ i-bound \ ts_{sb\cdot i}]

  have \ reads-consistent \ \text{False} \ O_{sb} \ m_{sb} \ sb .

  from \ reads-consistent-snoc-Ghost_{sb} \ [OF \ this]

  have \ reads-consistent \ \text{False} \ O_{sb} \ m_{sb} \ (sb \@ [\text{Ghost}_{sb} A L R W]).

  from \ valid-reads-nth-update \ [OF \ i-bound \ this]

  show \ ?thesis \ by \ (\text{simp \ add: \ ts_{sb\cdot j}' sb' \ O_{sb'}})

618
qed

have valid-sharing': valid-sharing $\mathcal{S}_{\text{sb}}' \mathcal{t}_{\text{sb}}'$
proof (intro-locales)
from outstanding-non-volatile-writes-unshared [OF i-bound $\mathcal{t}_{\text{sb}}$-i]
have non-volatile-writes-unshared $\mathcal{S}_{\text{sb}}$ (sb @ [Ghost $\mathcal{S}_{\text{sb}}$ A L R W])
by (auto simp add: non-volatile-writes-unshared-append)
from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared $\mathcal{S}_{\text{sb}}' \mathcal{t}_{\text{sb}}'$
by (simp add: $\mathcal{t}_{\text{sb}}' \mathcal{S}_{\text{sb}}'$)
next
from sharing-consis [OF i-bound $\mathcal{t}_{\text{sb}}$-i]
have consis': sharing-consistent $\mathcal{S}_{\text{sb}} \mathcal{O}_{\text{sb}}$ sb.
from A-shared-owned
have A \subseteq dom (share ?drop-sb $\mathcal{S}$) \cup acquired True sb $\mathcal{O}_{\text{sb}}$
by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write$\mathcal{S}_{\text{sb}}$)
moreover have dom (share ?drop-sb $\mathcal{S}$) \subseteq dom $\mathcal{S}$ \cup dom (share sb $\mathcal{S}_{\text{sb}}$)
proof
fix a'
assume a'-in: a' \in dom (share ?drop-sb $\mathcal{S}$)
from share-unshared-in [OF a'-in]
show a' \in dom $\mathcal{S}$ \cup dom (share sb $\mathcal{S}_{\text{sb}}$)
proof
assume a' \in dom (share ?drop-sb Map.empty)
from share-mono-in [OF this] share-append [of ?take-sb ?drop-sb]
have a' \in dom (share sb $\mathcal{S}_{\text{sb}}$)
by auto
thus ?thesis
by simp
next
assume a' \in dom $\mathcal{S}$ \land a' \notin all-unshared ?drop-sb
thus ?thesis by auto
qed

qed

ultimately
have A-subset: A \subseteq dom $\mathcal{S}$ \cup dom (share sb $\mathcal{S}_{\text{sb}}$) \cup acquired True sb $\mathcal{O}_{\text{sb}}$
by auto

have A \subseteq dom (share sb $\mathcal{S}_{\text{sb}}$) \cup acquired True sb $\mathcal{O}_{\text{sb}}$
proof --
{
fix x
assume x-A: x \in A
have x \in dom (share sb $\mathcal{S}_{\text{sb}}$) \cup acquired True sb $\mathcal{O}_{\text{sb}}$
proof --
{
assume x \in dom $\mathcal{S}$
from share-all-until-volatile-write-share-acquired [OF (sharing-consis $\mathcal{S}_{\text{sb}} \mathcal{t}_{\text{sb}}$) i-bound $\mathcal{t}_{\text{sb}}$-i this simplified $\mathcal{S}$]
A-unowned-by-others x-A

619
have ?thesis
  by (fastforce simp add: Let-def)
}
with A-subset show ?thesis using x-A by auto
qed
}
thus ?thesis by blast
qed

with consis' L-subset A-R R-acq
have sharing-consistent $S_{sb} \cup_{sb} (sb @ [\text{Ghost}_{sb} A L R W])$
  by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write_{sb})
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis $S_{sb}' \cap_{sb'} t_{sb}'$
  by (simp add: $t_{sb}' \cup_{sb'} sb' \cap_{sb} S_{sb}'$

next
from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts_{sb}-i]]
  show read-only-unowned $S_{sb}' \cap_{sb'} t_{sb}'$
  by (simp add: $S_{sb}' \cap_{sb'} O_{sb}' \cap_{sb} S_{sb}'$
next
from unowned-shared-nth-update [OF i-bound ts_{sb}-i subset-refl]
  show unowned-shared $S_{sb}' \cap_{sb'} t_{sb}'$
  by (simp add: $t_{sb}' \cup_{sb'} sb' \cap_{sb} O_{sb}' \cap_{sb} S_{sb}'$
next
from no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb}-i]
have no-write-to-read-only-memory $S_{sb} (sb @ [\text{Ghost}_{sb} A L R W])$
  by (simp add: no-write-to-read-only-memory-append)

from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
  show no-write-to-read-only-memory $S_{sb}' \cap_{sb'} t_{sb}'$
  by (simp add: $S_{sb}' \cup_{sb'} t_{sb}' \cap_{sb'} sb'$
qed

have tmps-distinct ': tmps-distinct ts_{sb}'
proof (intro-locales)
from load-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-load-tmps is_{sb}' by (simp add: is_{sb})
from load-tmps-distinct-nth-update [OF i-bound this]
  show load-tmps-distinct ts_{sb}' by (simp add: ts_{sb}')
next
from read-tmps-distinct [OF i-bound ts_{sb}-i]
have distinct-read-tmps (sb @ [\text{Ghost}_{sb} A L R W])
  by (auto simp add: distinct-read-tmps-append)
from read-tmps-distinct-nth-update [OF i-bound this]
  show read-tmps-distinct ts_{sb}' by (simp add: ts_{sb}' \cap_{sb'} sb')
next
from load-tmps-read-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' \cap_{sb'} read-tmps (sb @ [\text{Ghost}_{sb} A L R W]) = {}$
  by (auto simp add: read-tmps-append is_{sb})
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct ts\sb\ ′ by (simp add: ts\sb\ ′ sb′)
qed

have valid-sops′ \cdot valid-sops ts\sb\ ′
proof –
from valid-store-sops [OF i-bound ts\sb\ ′-i]
obtain
  valid-store-sops′ \forall sop\in\text{store-sops} is\sb\ ′, valid-sop sop
  by (auto simp add: is\sb\)
from valid-write-sops [OF i-bound ts\sb\ ′-i]
have valid-write-sops′ \forall sop\in\text{write-sops} (sb@ [\text{Ghost}_{\sb\ A L R W}]).
valid-sop sop
  by (auto simp add: write-sops-append)
from valid-sops-nth-update [OF i-bound valid-write-sops′ valid-store-sops′]
show ?thesis by (simp add: ts\sb\ ′ sb′)
qed

have valid-dd′ \cdot valid-data-dependency ts\sb\ ′
proof –
from data-dependency-consistent-instrs [OF i-bound ts\sb\ ′-i]
obtain
  dd-is: data-dependency-consistent-instrs (dom θ\sb\ ′) is\sb\ ′
by (auto simp add: θ\sb\)
from load-tmps-write-tmps-distinct [OF i-bound ts\sb\ ′-i]
have load-tmps is\sb\ ′ ∩ \bigcup (fst ' write-sops (sb@ [\text{Ghost}_{\sb\ A L R W}])) = {}
  by (auto simp add: write-sops-append)
from valid-data-dependency-nth-update [OF i-bound dd-is this]
show ?thesis by (simp add: ts\sb\ ′ sb′)
qed

have load-tmps-fresh′ \cdot load-tmps-fresh ts\sb\ ′
proof –
from load-tmps-fresh [OF i-bound ts\sb\ ′-i]
have load-tmps is\sb\ ′ ∩ dom θ\sb = {}
  by (auto simp add: is\sb)
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis by (simp add: ts\sb\ ′ θ\sb′)
qed

have enough-flushs′ \cdot enough-flushs ts\sb\ ′
proof –
from clean-no-outstanding-volatile-Write\sb [OF i-bound ts\sb\ ′-i]
have \neg D\sb \longrightarrow outstanding-refs is-volatile-Write\sb (sb@ [\text{Ghost}_{\sb\ A L R W}]) = {}
  by (auto simp add: outstanding-refs-append)
from enough-flushs-nth-update [OF i-bound this]
show ?thesis
  by (simp add: ts\sb\ ′ sb′ D\sb′)
qed

621
have valid-program-history' valid-program-history ts_{sb}:
proof -
from valid-program-history [OF i-bound ts_{sb-i}]
have causal-program-history is_{sb} sb .
then have causal': causal-program-history is_{sb}' (sb@[Ghost_{sb} A L R W])
  by (auto simp: causal-program-history-Ghost is_{sb})
from valid-last-prog [OF i-bound ts_{sb-i}]
have last-prog p_{sb} sb = p_{sb}.
hence last-prog p_{sb} (sb@[Ghost_{sb} A L R W]) = p_{sb}
  by (simp add: last-prog-append-Ghost_{sb})
from valid-program-history-nth-update [OF i-bound causal' this]
show ?thesis
  by (simp add: ts_{sb} sb'
qed

show ?thesis
proof (cases outstanding-refs is-volatile-Write_{sb} sb = {})
case True
from True have flush-all: takeWhile (Not ◦ is-volatile-Write_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
from True have suspend-nothing: dropWhile (Not ◦ is-volatile-Write_{sb}) sb = []
  by (auto simp add: outstanding-refs-conv)
hence suspends-empty: suspends = []
  by (simp add: suspends)
from suspends-empty is-sim have is =Ghost A L R W# is_{sb}'
  by (simp add: is_{sb})
with suspends-empty ts-i
have ts-i: tsli = (p_{sb}, Ghost A L R W# is_{sb}').
  \delta_{sb}'(), D, acquired True ?take-sb O_{sb},release ?take-sb (dom S_{sb}) R_{sb})
  by simp
from direct-memop-step.Ghost
have (Ghost A L R W# is_{sb}'),
  \delta_{sb}', ()\m, D, acquired True ?take-sb O_{sb},
    release ?take-sb (dom S_{sb}) R_{sb}, S) \rightarrow
  (is_{sb}'),
  \delta_{sb}(), \m, D, acquired True ?take-sb O_{sb} \cup A - R,
    augment-rels (dom S) R (release ?take-sb (dom S_{sb}) R_{sb}),
    S \oplus W R \ominus A L).
from direct-computation.concurrent-step.Memop [OF i-bound' ts-i this]
have (ts, m, S) \Rightarrow_d
  (ts[i := (p_{sb}, is_{sb}'),
    \delta_{sb}, (), D, acquired True ?take-sb O_{sb} \cup A - R,
    augment-rels (dom S) R (release ?take-sb (dom S_{sb}) R_{sb})))

622
m, S ⊕ W R ⊕ A L).

moreover

from suspend-nothing
have suspend-nothing': (dropWhile (Not ◦ is-volatile-Write{sb}) sb') = []
  by (simp add: sb')

have all-shared-A: ∀ p is O R D ⊕ sb. j < length ts_{sb} −→ i ≠ j −→
  ts_{sb} ! j = (p, is, ⊕, sb, D, O, R) −→
  all-shared (takeWhile (Not ◦ is-volatile-Write{sb}) sb) ∩ A = {}

proof −
{  
  fix j p j is j O j D j θ j sb j x
  assume j-bound: j < length ts_{sb}
  assume neq-i-j: i ≠ j
  assume jth: ts_{sb} ! j = (p_j, is_j, ⊕_j, sb_j, D_j, O_j, R_j)
  assume x-shared: x ∈ all-shared (takeWhile (Not ◦ is-volatile-Write{sb}) sb_j)
  assume x-A: x ∈ A
  have False
  proof −
    from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]
    have all-shared sb_j ⊆ all-acquired sb_j ∪ O_j
  moreover have all-shared (takeWhile (Not ◦ is-volatile-Write{sb}) sb_j) ⊆ all-shared sb_j
  using all-shared-append [of (takeWhile (Not ◦ is-volatile-Write{sb}) sb_j)
    (dropWhile (Not ◦ is-volatile-Write{sb}) sb_j)]
  by auto
  moreover
    from A-unaquired-by-others [rule-format, OF - neq-i-j] jth j-bound
    have A ∩ all-acquired sb_j = {} by auto
  moreover
    from A-unowned-by-others [rule-format, OF - neq-i-j] jth j-bound
    have A ∩ O_j = {}
  by (auto dest: all-shared-acquired-in)

  ultimately
  show False
  using x-A x-shared
  by blast
  qed
}
thus ?thesis by blast
qed
hence all-shared-L: \forall j \ p \is ORD \theta \sb. \ j < \text{length} \ ts_{sb} \rightarrow i \neq j \rightarrow \\
\text{ts}_{sb} ! j = (p, \is, \theta, \sb, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow \\
\text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap L = \{\}

using L-subset by blast

have all-shared-A: \forall j \ p \is ORD \theta \sb. \ j < \text{length} \ ts_{sb} \rightarrow i \neq j \rightarrow \\
\text{ts}_{sb} ! j = (p, \is, \theta, \sb, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow \\
\text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap A = \{\}

proof –

{ 
fix j p_j \is_j \mathcal{O}_j \mathcal{R}_j \mathcal{D}_j \emptyset_j \sb_j x 
assume j-bound: j < \text{length} \ ts_{sb} 
assume jth: ts_{sb}!j = (p_j, \is_j, \emptyset_j, \sb_j, \mathcal{D}_j, \mathcal{O}_j, \mathcal{R}_j) 
assume neq-i-j: i \neq j 
assume x-shared: x \in \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_j) 
assume x-A: x \in A 
have False 
proof –
from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]] 
have all-shared \sb_j \subseteq \text{all-acquired} \sb_j \cup \mathcal{O}_j, 

moreover have all-shared (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_j) \subseteq \text{all-shared} \sb_j 
using all-shared-append [of (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_j)] 
\ (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_j)] 
by auto 
moreover 
from A-unaquired-by-others [rule-format, OF - neq-i-j] jth j-bound 
have A \cap \text{all-acquired} \sb_j = \{\} by auto 
moreover 

from A-unowned-by-others [rule-format, OF - neq-i-j] jth j-bound 
have A \cap \mathcal{O}_j = \{\} 
by (auto dest: all-shared-acquired-in)

ultimately 
show False 
using x-A x-shared 
by blast 
qed 
}

thus \?thesis by blast 
qed 

hence all-shared-L: \forall j \ p \is ORD \mathcal{D} \emptyset \sb. \ j < \text{length} \ ts_{sb} \rightarrow i \neq j \rightarrow \\
\text{ts}_{sb} ! j = (p, \is, \emptyset, \sb, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow \\
\text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap L = \{\}

using L-subset by blast

have all-unshared-R: \forall j \ p \is ORD \mathcal{D} \emptyset \sb. \ j < \text{length} \ ts_{sb} \rightarrow i \neq j \rightarrow
\( \text{ts}_{sb} ! j = (p, \text{is}, \emptyset, sb, D, O, R) \longrightarrow \)
all-unshared (takeWhile \((\text{Not} \circ \text{is-volatile-Write}_{sb})\) \(sb\)) \(\cap R = \{\}\)

**proof**

\{ 
\begin{align*}
\text{fix } j & \ p_j \ \text{is}_j O_j R_j D_j \emptyset_j sb_j \ x \\
\text{assume } & \text{j-bound: } j < \text{length} \text{ts}_{sb} \\
\text{assume } & \text{neq-i-j: } i \neq j \\
\text{assume } & \text{jth: } \text{ts}_{sb} ! j = (p_j, \text{is}_j, \emptyset_j, sb_j, D_j, O_j, R_j) \\
\text{assume } & \text{x-unshared: } x \in \text{all-unshared} \ (\text{takeWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \\
\text{assume } & \text{x-R: } x \in R \\
\text{have } & \text{False} \\
\text{proof } \begin{array}{l}
\text{from } \text{unshared-acquired-or-owned } [\text{OF sharing-consis } [\text{OF j-bound jth}]] \\
\text{have } \text{all-unshared sb}_j \subseteq \text{all-acquired sb}_j \cup O_j, \\
\end{array}
\text{moreover have all-unshared} \ (\text{takeWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ sb_j) \subseteq \text{all-unshared sb}_j \\
\text{using all-unshared-append } [\text{of (takeWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \\
\ (\text{dropWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb)]] \\
\text{by auto} \\
\text{moreover} \\
\text{note ownership-distinct } [\text{OF i-bound j-bound neq-i-j ts}_{sb-i jth}]
\end{align*}
\}

ultimately

show False

using R-acq x-R x-unshared acquired-all-acquired [of True sb \(O_{sb}\)]

by blast

qed

} 

thus ?thesis by blast

qed

\{ 
\begin{align*}
\text{have all-acquired-R: } & \forall j \ p \ \text{is} \ O \ D \emptyset \ sb, j < \text{length} \text{ts}_{sb} \longrightarrow i \neq j \longrightarrow \\
\text{ts}_{sb} ! j = (p, \text{is}, \emptyset, sb, D, O, R) \longrightarrow \\
\text{all-acquired} \ (\text{takeWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap R = \{\} \\
\text{proof } \begin{array}{l}
\text{fix } j & \ p_j \ \text{is}_j O_j R_j D_j \emptyset_j sb_j \ x \\
\text{assume } & \text{j-bound: } j < \text{length} \text{ts}_{sb} \\
\text{assume } & \text{jth: } \text{ts}_{sb} ! j = (p_j, \text{is}_j, \emptyset_j, sb_j, D_j, O_j, R_j) \\
\text{assume } & \text{neq-i-j: } i \neq j \\
\text{assume } & \text{x-acq: } x \in \text{all-acquired} \ (\text{takeWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \\
\text{assume } & \text{x-R: } x \in R \\
\text{have } & \text{False} \\
\text{proof } \begin{array}{l}
\text{from } \text{x-acq have } x \in \text{all-acquired sb}_j \\
\text{using all-acquired-append } [\text{of takeWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \\
\ (\text{dropWhile} \ ((\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb)]
\end{array}
\end{array}
\end{align*}
\}
by auto
  moreover
  note ownership-distinct [OF i-bound j-bound neq-i-j ts$_{sb}$-i jth]
  ultimately
  show False
using R-acq x-R acquired-all-acquired [of True sb O$_{sb}$]
by blast
  qed
\}
thus $\exists$thesis by blast
  qed

have all-shared-R: $\forall j \ p \ is \ O \ R \ D \ @ \ sb. \ j < \ length \ ts_{sb} \longrightarrow i \neq j \longrightarrow$
  ts$_{sb}$ ! j = (p, is, @, sb, D, O, R) \rightarrow
  all-shared (takeWhile (Not \circ is-volatile-Write$_{sb}$) sb) \cap R = \{

proof --
\{
  fix \ j p j \ is \ j \ O \ j \ D \ j \ \theta \ j \ sb \ j \ x
  assume j-bound: \ j < \ length \ ts_{sb}
  assume jth: ts$_{sb}$ | j = (p$_j$, is$_j$, @, sb$_j$, D$_j$, O$_j$, R$_j$)
    assume neq-i-j: i \neq j
  assume x-shared: x \in all-shared (takeWhile (Not \circ is-volatile-Write$_{sb}$) sb$_j$)
  assume x-R: x \in R
  have False
  proof --
    from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]
    have all-shared sb$_j$ \subseteq all-acquired sb$_j$ \cup O$_j$,
    moreover have all-shared (takeWhile (Not \circ is-volatile-Write$_{sb}$) sb$_j$) \subseteq all-shared sb$_j$
    using all-shared-append [of (takeWhile (Not \circ is-volatile-Write$_{sb}$) sb$_j$)
      (dropWhile (Not \circ is-volatile-Write$_{sb}$) sb$_j$)]
    by auto
    moreover
      note ownership-distinct [OF i-bound j-bound neq-i-j ts$_{sb}$-i jth]
      ultimately
      show False
using R-acq x-R x-shared acquired-all-acquired [of True sb O$_{sb}$]
by blast
  qed
\}
thus $\exists$thesis by blast
  qed

note share-commute =
  share-all-until-volatile-write-append-Ghost$_{sb}$ [OF True \langle ownership-distinct ts$_{sb}$ \rangle
  (sharing-consis S$_{sb}$ ts$_{sb}$)
  i-bound ts$_{sb}$-i all-shared-L all-shared-A all-acquired-R all-unshared-R all-shared-R]
from $\mathcal{D}$

626
\( \text{have } D': D_{sb} = (D \lor \text{outstanding-refs is-volatile-Write}_{sb} (sb@[\text{Ghost}_{sb} A L R W]) \neq \{\}) \)

by (auto simp: outstanding-refs-append)

\( \text{have } \forall a \in R. \ (a \in (\text{dom share sb } S_{sb})) = (a \in \text{dom } S) \)

proof –

\( \{ \)

fix a

assume a-R: \( a \in R \)

have (a \in (\text{dom share sb } S_{sb})) = (a \in \text{dom } S)

proof –

from a-R R-acq acquired-all-acquired [of True sb O sb]

have \( a \in O_{sb} \cup \text{all-acquired sb} \)

by auto

from share-all-until-volatile-write-thread-local’ [OF ownership-distinct-ts sb]

sharing-consis-ts sb i-bound ts sb-i this] suspend-nothing

show \?thesis by (auto simp add: domIff S)

qed

\} 

then show \?thesis by auto

qed

from augment-rels-shared-exchange [OF this]

\( \text{have rel-commute: } \)

\( \text{augment-rels (dom } S) R (\text{release sb } (\text{dom } S_{sb}) R_{sb}) = \)

\( \text{release } (sb @ [\text{Ghost}_{sb} A L R W]) (\text{dom } S_{sb}') R_{sb} \)

by (clarsimp simp add: release-append S sb)

have \((ts_{sb}', m_{sb}, S_{sb}') \sim \\
\begin{align*}
(ts[i] & := (p_{sb}, i_{sb}, o_{sb}, d), \text{ acquired True } ?\text{take-sb } O_{sb} \cup A - R, \\
& \text{augment-rels (dom } S) R (\text{release } ?\text{take-sb } (\text{dom } S_{sb}) R_{sb})]), \\
m, S \oplus W R \ominus A L)
\end{align*}

apply (rule sim-config.intros)

apply (simp add: m ts sb' o sb' sb' o sb' share-commute)

using leq

apply (simp add: ts sb)

using i-bound i-bound' ts-sim ts-i True D'

apply (clarsimp simp add: Let-def nth-list-update

outstanding-refs-conv ts sb' O sb' R sb' S sb' o sb' sb' D sb' suspend-nothing' flush-all

rel-commute

acquired-append split: if-split-asm)

done

ultimately show ?thesis

using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct'
valid-sops'

627
valid-dd' load-tmeps-fresh' enough-flushs'
valid-program-history' valid' m_{sb}' S_{sb}' R_{sb}'
by auto
next
case False
then obtain r where r-in: r ∈ set sb and volatile-r: is-volatile-Write_{sb} r
by (auto simp add: outstanding-refs-conv)
from takeWhile-dropWhile-real-prefix
[OF r-in, of (Not ◦ is-volatile-Write_{sb}), simplified, OF volatile-r]
obtain a' v' sb'' A'' L'' R'' W'' sop' where
sb-split: sb = takeWhile (Not ◦ is-volatile-Write_{sb}) sb @ Write_{sb} True a' sop' v' A'' L'' R'' W''#
and
drop: dropWhile (Not ◦ is-volatile-Write_{sb}) sb = Write_{sb} True a' sop' v' A'' L'' R'' W''#
sb''
apply (auto)
subgoal for y ys
apply (case-tac y)
apply auto
done
done
from drop suspends have suspends: suspends = Write_{sb} True a' sop' v' A'' L'' R'' W''#
sb''
by simp
have (ts, m, S) ⇒ d^* (ts, m, S) by auto
moreover
have Write_{sb} True a' sop' v' A'' L'' R'' W'' ∈ set sb
by (subst sb-split) auto
note drop-app = dropWhile-append1
[OF this, of (Not ◦ is-volatile-Write_{sb}), simplified]

from takeWhile-append1 [where P=Not ◦ is-volatile-Write_{sb}, OF r-in] volatile-r
have takeWhile-app:
(takeWhile (Not ◦ is-volatile-Write_{sb}) (sb @ [Ghost_{sb} A L R W])) = (takeWhile (Not ◦ is-volatile-Write_{sb}) sb)
by simp

note share-commute = share-all-until-volatile-write-append-Ghost_{sb}' [OF False i-bound ts_{sb}-i]

from D
have D': D_{sb} = (D ∨ outstanding-refs is-volatile-Write_{sb} (sb@[Ghost_{sb} A L R W]) ∉ { })
by (auto simp: outstanding-refs-append)

have (ts_{sb}', m_{sb}', S_{sb}') ∼ (ts, m, S)
apply (rule sim-config.intros)

628
apply (simp add: m flush-all-until-volatile-write-append-Ghost-commute OF i-bound ts \( \ell \) i-] ! ts \( \ell \) O sb \( \vec{\ell} \) sb' )
apply (clarsimp simp add: S sb' ts sb' O sb' \( \vec{\ell} \) sb' share-commute)
using leq
apply (simp add: ts sb' \( \vec{\ell} \) sb'
using i-bound i-bound' ts-sim ts-i is-sim D'
apply (clarsimp simp add: Let-def nth-list-update is-sim drop-app read-tmps-append suspends
prog-instrs-append-Ghost sb instrs-append-Ghost sb hd-prog-append-Ghost sb
drop is sb ts sb' sb' O sb' \( \vec{\ell} \) sb R sb' S sb' \( \vec{\ell} \) sb' D sb' takeWhile-app split: if-split-asm)
done
ultimately show ?thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' valid-dd'
valid-sops' load-tmps-fresh' enough-flushs'
valid-program-history' valid' m sb' S sb'
by (auto simp del: fun-upd-apply )
qed
qed
next
case (StoreBuffer i p sb is sb \( \vec{\ell} \) sb sb D sb O sb R sb sb'
then obtain
t sb' \( \vec{\ell} \) sb' = ts \( \ell \) [i := (p sb, is sb, \( \vec{\ell} \) sb', D sb, O sb', R sb')]
and
i-bound: i < length ts sb
and
ts \( \ell \) i-: ts \( \ell \) ! i = (p sb, is sb, \( \vec{\ell} \) sb, sb, D sb, O sb, R sb)
and
flush: (m sb sb, O sb, R sb, S sb) \( \vec{\ell} \) i
by auto
from sim obtain
m: m = flush-all-until-volatile-write ts sb m sb and
S: S = share-all-until-volatile-write ts sb S sb and
leq: length ts sb = length ts and
ts-sim: \( \forall i < \) length ts sb.
let (p, is sb, \( \vec{\ell} \), sb, D sb, O sb, R)
= ts sb ! i;
suspends = dropWhile (Not \( \circ \) is-volatile-Write sb) sb
in \( \exists \) is D. instrs suspends \( @ \) is sb = is \( @ \) prog-instrs suspends \( \wedge \)
D sb = (D \( \vee \) outstanding-refs is-volatile-Write sb sb \( \neq \) {\}) \( \wedge \)
ts ! i =
(hd-prog p suspends,
is,
\( \vec{\ell} \mid \) (dom \( \vec{\ell} \) - read-tmps suspends), (),
D,
acquired True (takeWhile (Not \( \circ \) is-volatile-Write sb) sb) O sb,
release (takeWhile (Not \( \circ \) is-volatile-Write sb) sb) (dom S sb) R)
by cases blast
from i-bound leq have i-bound': i < length ts
by auto
have split-sb: sb = takeWhile (Not ∘ is-volatile-Write_{sb}) sb @ dropWhile (Not ∘ is-volatile-Write_{sb}) sb
(is sb = ?take-sb@?drop-sb)
by simp

from ts-sim [rule-format, OF i-bound] ts_{sb-i} obtain suspends is \mathcal{D} where
suspends: suspends = dropWhile (Not ∘ is-volatile-Write_{sb}) sb and
is-sim: instrs suspends @ is_{sb} = is @ prog-instrs suspends and
\mathcal{D}: \mathcal{D}_{sb} = (\mathcal{D} ∨ outstanding-refs is-volatile-Write_{sb} sb ≠ \{\}) and
ts-i: ts ! i =
(hd-prog p_{sb} suspends, is,
\vartheta_{sb} | (dom \vartheta_{sb} − read-tmps suspends), (),\mathcal{D}, acquired True ?take-sb \mathcal{O}_{sb},
release ?take-sb (dom S_{sb}) \mathcal{R}_{sb})
by (auto simp add: Let-def)

from flush-step-preserves-valid [OF i-bound ts_{sb-i} flush valid]
have valid': valid ts_{sb}'
by (simp add: ts_{sb}')

from flush obtain r where sb: sb=r#sb'
by (cases) auto

from valid-history [OF i-bound ts_{sb-i}]
have history-consistent \vartheta_{sb} (hd-prog p_{sb} sb) sb.
then
have hist-consis': history-consistent \vartheta_{sb} (hd-prog p_{sb} sb') sb'
by (auto simp add: sb intro: history-consistent-hd-prog
split: memref.splits option.splits)
from valid-history-nth-update [OF i-bound this]
have valid-hist': valid-history program-step ts_{sb}' by (simp add: ts_{sb}')

from read-tmps-distinct [OF i-bound ts_{sb-i}]
have dist-sb': distinct-read-tmps sb'
by (simp add: sb split: memref.splits)

have tmps-distinct': tmps-distinct ts_{sb}'
proof (intro-locales)
from load-tmps-distinct [OF i-bound ts_{sb-i}]
have distinct-load-tmps is_{sb}.

from load-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-distinct ts_{sb}'
by (simp add: ts_{sb}')
next
from read-tmps-distinct-nth-update [OF i-bound dist-sb']
show read-tmps-distinct ts_{sb}'
by (simp add: ts_{sb}')
next
from load-tmps-read-tmps-distinct [OF i-bound ts_{sb-i}]

630
have load-tmps is_{sb} ∩ read-tmps sb' = {}
by (auto simp add: sb split: memref.splits)
from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]
show load-tmps-read-tmps-distinct ts_{sb}' by (simp add: ts_{sb}')
qed

from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb} ∩ Union (fst ' write-sops sb') = {}
by (auto simp add: sb split: memref.splits)
from valid-data-dependency-nth-update
[OF i-bound data-dependency-consistent-instrs [OF i-bound ts_{sb}-i] this]
have valid-dd': valid-data-dependency ts_{sb}'
by (simp add: ts sb')

from valid-store-sops [OF i-bound ts_{sb}-i] valid-write-sops [OF i-bound ts_{sb}-i]
valid-sops-nth-update [OF i-bound]
have valid-sops': valid-sops ts_{sb}'
by (cases r) (auto simp add: sb ts sb')

have load-tmps-fresh': load-tmps-fresh ts_{sb}'
proof –
from load-tmps-fresh [OF i-bound ts_{sb}-i]
have load-tmps is_{sb} ∩ dom θ_{sb} = {}. 
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}')
qed

have enough-flushs': enough-flushs ts_{sb}'
proof –
from clean-no-outstanding-volatile-Write_{sb} [OF i-bound ts_{sb}-i]
have ¬D_{sb} → outstanding-refs is-volatile-Write_{sb} sb' = {}
by (auto simp add: if-split-asm)
from enough-flushs-nth-update [OF i-bound this]
show ?thesis
by (simp add: ts_{sb}' sb)
qed

show ?thesis
proof (cases r)
case (Write_{sb} volatile a sop v A L R W)
from flush this
have m_{sb}' : m_{sb}' = (m_{sb}(a := v))
by cases (auto simp add: sb)

have non-volatile-owned: ¬ volatile → a ∈ O_{sb}
proof (cases volatile)
case True thus ?thesis by simp
next
case False
with outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb}-i]
have \( a \in O_{sb} \)
by (simp add: sb Write_{sb})
thus \(?thesis\) by simp
qed

have a-unowned-by-others:
\( \forall j < \text{length ts}_{sb} \ldotp \ i \neq j \rightarrow (\text{let } (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \text{ts}_{sb} ! j \in a \notin O_j \cup \text{all-acquired sb}_j) \)
proof (unfold Let-def, clarify del: notI)
fix \( j \ p_j \ i_{sbj} \ O_j \ D_j \ \notj_{sbj} \)
assume j-bound: \( j < \text{length ts}_{sb} \)
assume neq: \( i \neq j \)
assume ts-j: \( \text{ts}_{sb} ! j = (p_j,i_{sbj},\notj_{sbj},D_j,O_j,R_j) \)
show a \( \notin O_j \cup \text{all-acquired sb}_j \)
proof (cases volatile)
case True
from outstanding-volatile-writes-unowned-by-others [OF i-bound j-bound neq ts_{sb}-i ts-j]
show \(?thesis\)
by (simp add: sb Write_{sb} True)
next
case False
with non-volatile-owned
have a \( \in O_{sb} \)
by simp
with ownership-distinct [OF i-bound j-bound neq ts_{sb}-i ts-j]
show \(?thesis\)
by blast
qed
qed

from valid-reads [OF i-bound ts_{sb}-i]
have reads-consis: reads-consistent False \( O_{sb} \) \( m_{sb} \) sb .
by auto
with non-volatile-owned-or-read-only-outstanding-non-volatile-writes
[OF outstanding-non-volatile-refs-owned-or-read-only [OF j-bound ts_{sb}\text{-}j]]
have a \in \mathcal{O}_j \cup \text{all-acquired sb}_j
by auto
with a-unowned-by-others [rule-format, OF j-bound neq-i-j] ts_{sb}\text{-}j
show False
by auto
qed

note a-notin-others = this

from a-notin-others
have a-notin-others':
\forall j < \text{length ts}_{sb}. i \neq j \rightarrow
(let (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = ts\text{-}j in a \notin \text{outstanding-refs is-Write}_{sb} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j))
by (fastforce simp add: \text{Let-def})
obtain D f where sop: sop=(D,f) by (cases sop) auto
from \text{valid-history} [OF i-bound ts_{sb}\text{-}i] sop sb \text{ Write}_{sb}
obtain D\text{-}tmps: D \subseteq \text{dom} \vartheta_{sb} \text{ and } f\text{-v: } f \vartheta_{sb} = v \text{ and } D\text{-sb}' : D \cap \text{read-tmps sb}' = \{\}
by (auto simp add: \text{sb} \text{ Write}_{sb})

let \vartheta = (\vartheta_{sb} \mid (\text{dom} \vartheta_{sb} - \text{read-tmps sb}'))
from D\text{-}tmps D\text{-sb}'
have D\text{-tmps}': D \subseteq \text{dom} \vartheta
by auto
from \text{valid-write-sops} [OF i-bound ts_{sb}\text{-}i, rule-format, of sop]
have valid-sop sop
by (auto simp add: \text{sb} \text{ Write}_{sb})
from this [simplified sop]
interpret valid-sop (D,f) .
from D\text{-tmps D\text{-sb}'}
have ((\text{dom} \vartheta_{sb} - \text{read-tmps sb}') \cap D) = D
by blast
with valid-sop [OF refl D\text{-tmtps}] valid-sop [OF refl D\text{-tmps}'] f\text{-v}
have f\text{-v}': f ?\vartheta = v
by auto

have valid-program-history': valid-program-history ts_{sb}'
proof
from valid-program-history [OF i-bound ts_{sb}\text{-}i]
have causal-program-history is_{sb} sb .
then have causal': causal-program-history is_{sb} sb'
by (simp add: \text{sb} \text{ Write}_{sb} causal-program-history-def)

633
from valid-last-prog [OF i-bound ts_{sb}\text{-i}]
have last-prog p_{sb} sb = p_{sb}.
hence last-prog p_{sb} sb' = p_{sb}
  by (simp add: sb Write_{sb})

from valid-program-history-nth-update [OF i-bound causal' this]
show ?thesis
  by (simp add: ts_{sb}')
qed

show ?thesis
proof (cases volatile)
case True
note volatile = this
from flush Write_{sb} volatile
obtain
  O_{sb}': O_{sb} = O_{sb} \cup A - R \text{ and }
  S_{sb}': S_{sb} = S_{sb} \oplus W R \oplus A L \text{ and }
  R_{sb}': R_{sb} = Map.empty
  by cases (auto simp add: sb)

from sharing-consis [OF i-bound ts_{sb}\text{-i}]
obtain
  A-shared-owned: A \subseteq \text{dom } S_{sb} \cup O_{sb} \text{ and }
  L-subset: L \subseteq A \text{ and }
  A-R: A \cap R = \{\} \text{ and }
  R-owned: R \subseteq O_{sb}
  by (clarsimp simp add: sb Write_{sb} volatile)

from sb Write_{sb} True have take-empty: takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb = []
  by (auto simp add: outstanding-refs-conv)
from sb Write_{sb} True have suspend-all: dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
hence suspends-all: suspends = sb
  by (simp add: suspends)
from is-sim
have is-sim: Write True a (D, f) A L R W# instrs sb' @ is_{sb} = is @ prog-instrs sb'
  by (simp add: True Write_{sb} suspends-all sb sop)
from valid-program-history [OF i-bound ts_{sb}\text{-i}]
interpret causal-program-history is_{sb} sb.
from valid-last-prog [OF i-bound ts_{sb}\text{-i}]
have last-prog: last-prog p_{sb} sb = p_{sb}.
from causal-program-history [of [Write _sb True a (D, f) v A L R W] sb] is-sim
obtain is' where
is: is = Write True a (D, f) A L R W# is' and
is'sim: instrs sb @ is_ssb = is' @ prog-instrs sb'
by (auto simp add: sb Write sb volatile sop)

from causal-program-history have
causal-program-history-sb': causal-program-history is_{sb} sb'
apply
apply (rule causal-program-history.intro)
apply (auto simp add: sb Write sb)
done

from ts-i have ts-i: ts ! i =
  (hd-prog p_{sb} sb', Write True a (D, f) A L R W# is', {?θ, ()}, D, acquired True
  ?take-sb O_{sb},
  release ?take-sb (dom S_{sb}) R_{sb})
by (simp add: suspends-all sb Write sb is)

let ?ts'i := ts[i := (hd-prog p_{sb} sb', is', {?θ, ()}, True, acquired True ?take-sb O_{sb} ∪ A − R,
Map.empty)]

from i-bound' have ts'i: ts[i := (hd-prog p_{sb} sb', is', {?θ, ()}, True, acquired True ?take-sb
O_{sb} ∪ A − R, Map.empty)
by simp

from no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb}-i]
have a-not-ro: a ∉ read-only S_{sb}
by (clarsimp simp add: sb Write sb volatile)

{ fix j
  fix p_j is_{sbj} O_j R_j D_{sbj} θ_j sb_j
  assume j-bound: j < length ts_{sb}
  assume ts_{sb}-j: ts_{sb}[j := (p_j, is_{sbj}, θ_j, sb_j, D_{sbj}, O_j, R_j)]
  assume neq-i-j: i≠j
  have a ∉ unforwarded-non-volatile-reads (dropWhile (Not ◦ is-volatile-Write sb) sb_j) {} proof
  let ?take-sbj = takeWhile (Not ◦ is-volatile-Write sb) sb_j
  let ?drop-sbj = dropWhile (Not ◦ is-volatile-Write sb) sb_j
obtain a-unowned: a ∉ O_j and a-unacq: a ∉ all-acquired sb_j
by auto

with all-acquired-append [of ?take-sbj ?drop-sbj] acquired-takeWhile-non-volatile-Write sb [of sb_j O_j]
have a-unacq-take: a ∉ acquired True ?take-sb_j O_j
by (auto simp add: )

635
\textbf{note} nvo-j = outstanding-non-volatile-refs-owned-or-read-only \{OF j-bound ts_{sb}-j\}

\textbf{from} non-volatile-owned-or-read-only-drop \{OF nvo-j\}
\textbf{have} nvo-drop-j: non-volatile-owned-or-read-only True (share \?take-sb \_j \_S_{sb})
\hspace{1cm} (acquired True \?take-sb \_j \_O_j) \?drop-sb\_j.

\textbf{note} consis-j = sharing-consist \{OF j-bound ts_{sb}-j\}
\textbf{with} sharing-consistent-append \{of \_S_{sb} \_j \_take-sb \_j \_O_j \_j\}
\textbf{obtain} consis-take-j: sharing-consistent \_S_{sb} \_j \_take-sb \_j \_O_j \_j and
\hspace{1cm} consis-drop-j: sharing-consistent \{share \?take-sb \_j \_S_{sb}\}
\hspace{1cm} (acquired True \?take-sb \_j \_O_j) \?drop-sb\_j
\hspace{1cm} \textbf{by auto}

\textbf{from} in-unforwarded-non-volatile-reads-non-volatile-Read_{sb} \{OF a-in\}
\textbf{have} a-in': a \in outstanding-refs is-non-volatile-Read_{sb} ?drop-sb\_j.

\textbf{note} reads-consis-j = valid-reads \{OF j-bound ts_{sb}-j\}
\textbf{from} reads-consistent-drop \{OF this\}
\textbf{have} reads-consis-drop-j:
\hspace{1cm} reads-consistent True (acquired True \?take-sb \_j \_O_j) (flush \?take-sb \_j \_m_{sb}) \?drop-sb\_j.

\textbf{from} read-only-share-all-shared \{of \?take-sb \_j \_S_{sb}\} a-not-ro
\hspace{1cm} all-shared-acquired-or-owned \{OF consis-take-j\}
\hspace{1cm} all-acquired-append \{of \?take-sb \_j \?drop-sb\_j\} a-unowned a-unacq
\textbf{have} a-not-ro-j: a \notin read-only (share \?take-sb \_j \_S_{sb})
\hspace{1cm} \textbf{by auto}

\textbf{from} ts-sim \{rule-format, OF j-bound\} ts_{sb}-j j-bound
\textbf{obtain} suspends_j is_j \_D_{j} \_R_{j} \textbf{where}
\hspace{1cm} suspends_j: suspends_j = \?drop-sb\_j and
\hspace{1cm} is_j: instrs suspends_j @ is_{sbj} = is_j @ prog-instrs suspends_j and
\hspace{1cm} \_D_{j}: \_D_{sbj} = (\_D_{j} \lor \text{outstanding-refs is-volatile-Write}_{sb}\_j \neq \{\}) and
\hspace{1cm} ts_{j}: ts_{j} = (hd-prog p_{j} suspends_{j}, is_{j},
\hspace{1cm} \_j \mid' (dom \_j - \text{read-tmps suspends}_{j}),(),
\hspace{1cm} \_D_{j}, \text{acquired True \?take-sb \_j \_O_j,\_R_j})
\hspace{1cm} \textbf{by (auto simp: Let-def)}

\textbf{from} valid-last-prog \{OF j-bound ts_{sb}-j\} \textbf{have} last-prog: last-prog p_{j} sb_{j} = p_{j}.

\textbf{from} j-bound i-bound' leq \textbf{have} j-bound-ts': j < length ts
\hspace{1cm} \textbf{by simp}
\hspace{1cm} \textbf{from} read-only-read-acquired-unforwarded-acquire-witness \{OF nvo-drop-j\}
\hspace{1cm} consis-drop-j
\hspace{1cm} a-not-ro-j a-unacq-take a-in]
have False

proof

assume \( \exists \) sop \( a' \) v ys zs A L R W.

?drop-sb\(_j\) = ys @ Write\(_s\)b True \( a' \) sop v A L R W \# zs \& a \in A \&
a \notin outstanding-refs is-Write\(_s\)b ys \& a' \neq a

with suspends\(_j\)

obtain \( a' \) sop' v' ys zs' A' L' R' W' where

split-suspends\(_j\); suspends\(_j\) = ys @ Write\(_s\)b True \( a' \) sop' v' A' L' R' W'\# zs' (is suspends\(_j\)=?suspends) and

a-A'; a \in A' and

no-write: a \notin outstanding-refs is-Write\(_s\)b (ys @ Write\(_s\)b True \( a' \) sop' v' A' L' R' W')

by (auto simp add: outstanding-refs-append)

from last-prog

have lp: last-prog p\(_j\) suspends\(_j\) = p\(_j\)

apply –

apply (rule last-prog-same-append [where sb=?take-sb\(_j\)])

apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))

apply simp

done

from sharing-consis [OF j-bound ts\(_s\)b-j]

have sharing-consis-j: sharing-consistent \( S_{sb} \) O\(_j\) sb\(_j\).

then have A' L' R' A' \cap R' = \{\}


from valid-program-history [OF j-bound ts\(_s\)b-j]

have causal-program-history is\(_s\)b\(_j\) sb\(_j\).

then have cph: causal-program-history is\(_s\)b\(_j\) ?suspends

apply –

apply (rule causal-program-history-suffix [where sb=?take-sb\(_j\)] )

apply (simp only: split-suspends\(_j\) [symmetric] suspends\(_j\))

apply (simp add: split-suspends\(_j\))

done

from valid-reads [OF j-bound ts\(_s\)b-j]

have reads-consis-j: reads-consistent False O\(_j\) m\(_s\)b sb\(_j\).

from reads-consis-j: reads-consistentFalse O\(_j\) m\(_s\)b sb\(_j\).

from reads-consis-flush-all-until-volatile-write [OF valid-ownership-and-sharing S\(_s\)b ts\(_s\)b]

j-bound ts\(_s\)b-j this]

have reads-consis-m-j: reads-consistent True (acquired True ?take-sb\(_j\) O\(_j\)) m suspends\(_j\)

by (simp add: m suspends\(_j\))

hence reads-consis-ys: reads-consistent True (acquired True ?take-sb\(_j\) O\(_j\)) m (ys@[Write\(_s\)b True \( a' \) sop' v' A' L' R' W'])

by (simp add: split-suspends\(_j\) reads-consistent-append)

from valid-write-sops [OF j-bound ts\(_s\)b-j]
have $\forall \text{sop} \in \text{write-sops} \ (\text{?take-sb}_j \ ?\text{suspends})$. valid-sop sop
by (simp add: split-suspends$_j$ [symmetric] suspends$_j$)
then obtain valid-sops-take: $\forall \text{sop} \in \text{write-sops} \ ?\text{take-sb}_j$. valid-sop sop and
valid-sops-drop: $\forall \text{sop} \in \text{write-sops} \ (\text{ys}@[\text{Write}_s \text{b} \ True \ a' \ \text{sop}' \ v' \ A' \ L' \ R' \ W'])$. valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound ts$_{sb-j}$]
have distinct-read-tmps (?take-sb$_j$ ?suspends$_j$)
by (simp add: split-suspends$_j$ [symmetric] suspends$_j$)
then obtain
read-tmps-take-drop: read-tmps ?take-sb$_j$ \cap read-tmps suspends$_j$ = {}
and
distinct-read-tmps-drop: distinct-read-tmps suspends$_j$
apply (simp only: split-suspends$_j$ [symmetric] suspends$_j$)
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound ts$_{sb-j}$]
have h-consis:
history-consistent $\theta_j$ (hd-prog $p_j$ (?take-sb$_j$ ?suspends$_j$)) (?take-sb$_j$ ?suspends$_j$)
apply (simp only: split-suspends$_j$ [symmetric] suspends$_j$)
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog $p_j$ sb$_j$) ?take-sb$_j$ = (hd-prog $p_j$ suspends$_j$)
proof –
from last-prog have last-prog $p_j$ (?take-sb$_j$ ?drop-sb$_j$) = $p_j$
by simp
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog $p_j$ suspends$_j$) ?take-sb$_j$ = hd-prog $p_j$ suspends$_j$
by (simp only: split-suspends$_j$ [symmetric] suspends$_j$)
moreover
have last-prog (hd-prog $p_j$ (?take-sb$_j$ @ suspends$_j$)) ?take-sb$_j$ =
last-prog (hd-prog $p_j$ suspends$_j$) ?take-sb$_j$
apply (simp only: split-suspends$_j$ [symmetric] suspends$_j$)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspends$_j$ [symmetric] suspends$_j$)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
h-consis] last-prog-hd-prog
have hist-consis': history-consistent $\theta_j$ (hd-prog $p_j$ suspends$_j$) suspends$_j$
by (simp add: split-suspends$_j$ [symmetric] suspends$_j$)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-j]
have no-vol-read: outstanding-refs is-volatile-Read$_s_b$
(ys@[Write$_s_b$ True a' sop' v' A' L' R' W']) = {}

638
by (auto simp add: outstanding-refs-append splitsn [symmetric]
        split-suspends)

have acq-simp:
  acquired True (ys @ [Write \sb True a' sop' v' A' L' R' W])
  (acquired True ?take-sb j Oj) =
  acquired True ys (acquired True ?take-sb j Oj) ∪ A' − R'
by (simp add: acquired-append)

from flush-store-buffer-append [where \sb=ys@[Write \sb True a'
   sop' v' A' L' R' W] and \sb'=zs', simplified,
  OF j-bound-ts'isj [simplified split-suspends] cph [simplified suspends]
  tsj [simplified split-suspends]
  refl lp [simplified split-suspends] reads-consis
  hist-consis' [simplified split-suspends] valid-sops-drop
  distinct-read-tmps-drop [simplified split-suspends]
  no-volatile-Read\sb-volatile-reads-consistent [OF no-vol-read], where
  S=S

obtain isj' \Rj' where
  isj'': instrs zs' @ is\sbj = isj' @ prog-instrs zs' and
  steps-ys: (ts, m, S) ⇒ \d* (ts[j:=(last-prog
    (hd-prog p j (Write \sb True a' sop' v' A' L' R' W'# zs'))
    (ys@[Write \sb
    True a' sop' v' A' L' R' W']))
    isj',
    \dj' | (dom \dj' − read-tmps zs'),
    ((), True, acquired True ys (acquired True ?take-sb j Oj) ∪ A' − R',\Rj')),
  flush (ys@[Write \sb True a' sop' v' A' L' R' W]) m,
  share (ys@[Write \sb True a' sop' v' A' L' R' W]) S)
  (is (\_,\_,\_) ⇒ \d* (?ts-ys,?m-ys,?shared-ys))
by (auto simp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: i < length ?ts-ys
by auto

from i-bound' neq-i-j ts-i
have ts-ys-i: ?ts-ys!i = (hd-prog p\sb sb', Write True a (D, f) A L R W# is', ?, (),
  D),
  acquired True ?take-sb O\sb,release ?take-sb (dom S\sb) \R\sb)
by simp

note conflict-computation = steps-ys

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe: safe-delayed (?ts-ys,?m-ys,?shared-ys).

with safe-delayedE [OF safe i-bound-ys ts-ys-i]
have a-unowned:
\( \forall j < \text{length } ?ts-ys \cdot i \neq j \longrightarrow (\text{let } (O_j) = \text{map owned } ?ts-ys!j \text{ in } a \notin O_j) \)

**apply** cases

**apply** (auto simp add: Let-def sb)

**done**

**from** a-A' a-unowned [rule-format, of j] neq-i-j j-bound leq A'-R'

**show** False

**by** (auto simp add: Let-def)

**next**

**assume** \( \exists A L R W \ ys \ zs \cdot \text{?drop-sb}_j = \ys @ \text{Ghost}_{sb} A L R W\# zs \land a \in A \land a \notin \text{outstanding-refs is-Write}_{sb} \ ys \) with 

**obtain** \( \ys \ zs' A' L' R' W' \ where \)

split-suspendsj: suspendsj = \( \ys @ \text{Ghost}_{sb} A' L' R' W'\# zs' \) (is suspendsj=?suspends)

**and**

a-A': a \in A' and

no-write: a \notin \text{outstanding-refs is-Write}_{sb} (\ys @ [\text{Ghost}_{sb} A' L' R' W'])

**by** (auto simp add: outstanding-refs-append)

**from** last-prog

**have** lp: last-prog \( p_j \) suspendsj = \( p_j \)

**apply** −

**apply** (rule last-prog-same-append [where sb=?take-sb]\( j \))

**apply** (simp only: split-suspendsj [symmetric] suspendsj)

**apply** simp

**done**

**from** valid-program-history [OF j-bound ts_{sb-j}]

**have** causal-program-history is_{sbj} sbj.

**then have** cph: causal-program-history is_{sbj} ?suspends

**apply** −

**apply** (rule causal-program-history-suffix [where sb=?take-sb]\( j \) )

**apply** (simp only: split-suspendsj [symmetric] suspendsj)

**apply** (simp add: split-suspendsj)

**done**

**from** valid-reads [OF j-bound ts_{sb-j}]

**have** reads-consis-j: reads-consistent False \( O_j \) \( m_{sb} \) sbj.

**from** reads-consistent-flush-all-until-volatile-write [OF (valid-ownership-and-sharing 

\( S_{sb} \) ts_{sb-j} j-bound ts_{sb-j} this]

**have** reads-consis-m-j: reads-consistent True (acquired True \( ?take-sb_j O_j \) ) \( m \) suspendsj

**by** (simp add: m suspendsj)

**hence** reads-consis-ys: reads-consistent True (acquired True \( ?take-sb_j O_j \) ) \( m \) \( (\ys@[\text{Ghost}_{sb} A' L' R' W']) \)

**by** (simp add: split-suspendsj reads-consistent-append)
from valid-write-sops [OF j-bound ts\sb{-}j]

have \( \forall \text{sop} \in \text{write-sops} \ (\text{?take-sb@?suspends}) \). valid-sop sop

by (simp add: split-suspends\sb{j} [symmetric] suspends\sb{j})

then obtain valid-sops-take: \( \forall \text{sop} \in \text{write-sops} \ (\text{?take-sb}) \). valid-sop sop and valid-sops-drop: \( \forall \text{sop} \in \text{write-sops} \ (ys@[\text{Ghost}\sb{\sb{\sb{\sb{\sb{A'}}\ \ \ R'}\ \ W}}]). \) valid-sop sop

apply (simp only: write-sops-append)

apply auto

done

from read-tmps-distinct [OF j-bound ts\sb{-}j]

have \( \text{distinct-read-tmps} \ (\text{?take-sb@suspends}) \)

by (simp add: split-suspends\sb{j} [symmetric] suspends\sb{j})

then obtain read-tmps-take-drop: \( \text{read-tmps} \cap \text{suspends} = \{\} \) and distinct-read-tmps-drop: \( \text{distinct-read-tmps} \)

apply (simp only: split-suspends\sb{j} [symmetric] suspends\sb{j})

apply (simp only: distinct-read-tmps-append)

done

from valid-history [OF j-bound ts\sb{-}j]

have h-consis:

\( \text{history-consistent} \ \theta\sb{j} \ (\text{hd-prog} \ p\sb{j} \ (\text{?take-sb@?drop-sb})) \ (\text{?take-sb@suspends}) \)

apply (simp only: split-suspends\sb{j} [symmetric] suspends\sb{j})

done

from sharing-consis [OF j-bound ts\sb{-}j]

have sharing-consis-j: \( \text{sharing-consistent} \ \mathcal{S}\sb{j}\sb{O}\sb{j}\sb{j} \).

then have \( A' R' : A' \cap R' = \{\} \)

by (simp add: sharing-consistent-append [of - - ?take-sb ?drop-sb, simplified]

suspends\sb{j} [symmetric] split-suspends\sb{j} sharing-consistent-append)

have last-prog-hd-prog: \( \text{last-prog} \ (\text{hd-prog} \ p\sb{j} \ \text{sb}) \ ?\text{take-sb} = (\text{hd-prog} \ p\sb{j} \ \text{suspends}) \)

proof |

from last-prog have \( \text{last-prog} \ p\sb{j} \ (\text{?take-sb@?drop-sb}) = p\sb{j} \)

by simp

from last-prog-hd-prog-append' [OF h-consis] this

have \( \text{last-prog} \ (\text{hd-prog} \ p\sb{j} \ \text{suspends}) \ ?\text{take-sb} = \text{hd-prog} \ p\sb{j} \ \text{suspends} \)

by (simp only: split-suspends\sb{j} [symmetric] suspends\sb{j})

moreover

have \( \text{last-prog} \ (\text{hd-prog} \ p\sb{j} \ (\text{?take-sb@suspends})) \ ?\text{take-sb} = \text{last-prog} \ (\text{hd-prog} \ p\sb{j} \ \text{suspends}) \ ?\text{take-sb} \)

apply (simp only: split-suspends\sb{j} [symmetric] suspends\sb{j})

by (rule last-prog-hd-prog-append)

ultimately show \( ?\text{thesis} \)

by (simp add: split-suspends\sb{j} [symmetric] suspends\sb{j})

qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis] last-prog-hd-prog

641
have hist-cons: history-consistent \( \theta_j \) (hd-prog \( p_j \) suspends\(_j\)) suspends\(_j\)
by (simp add: split-suspends\(_j\) [symmetric] suspends\(_j\))
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis-j]
have no-vol-read: outstanding-refs is-volatile-Read\(_sb\)
(ys@[Ghost\(_sb\) A' L' R' W']) = {}
by (auto simp add: outstanding-refs-append suspends\(_j\) [symmetric]
split-suspends\(_j\))

have acq-simp:
acquired True (ys @ [Ghost\(_sb\) A' L' R' W']
(acquired True ?take-sb\(_j\) O\(_j\)) =
acquired True ys (acquired True ?take-sb\(_j\) O\(_j\)) \( \cup \) A' \( \setminus \) R'
by (simp add: acquired-append)

from flush-store-buffer-append [where sb=ys@[Ghost\(_sb\) A' L' R' W'] and sb'=zs', simplified,
OF j-bound-ts' is\(_j\) [simplified split-suspends\(_j\)] cph [simplified suspends\(_j\)]
ts\(_j\) [simplified split-suspends\(_j\)] refl lp [simplified split-suspends\(_j\)] reads-consis-ys
hist-consis' [simplified split-suspends\(_j\)] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspends\(_j\)] no-volatile-Read\(_sb\)-volatile-reads-consistent [OF no-vol-read], where
\( S=S'\)

obtain is\(_j\)' \( \mathcal{R}_j'\) where
is\(_j\)' : instrs zs' @ is\(_sbj\) = is\(_j\)' @ prog-instrs zs' and
steps-ys: (ts, m,\( S\)) \( \Rightarrow \) \( d^*\)
(ts[\( j\)]:=last-prog
(\( (\text{hd-prog } p_j (\text{Ghost}_{sb} A' L' R' W'\# zs') \text{, } ys@[\text{Ghost}_{sb} A' L' R' W']),
\( \theta_j'\),
(\( j\) | (dom \( \theta_j \) \( \setminus \) read-tmps zs'),
(),
\( D_j\) \( \cup \) outstanding-refs is-volatile-Write\(_sb\) ys \( \neq \) {}
(acquired True ?take-sb\(_j\) O\(_j\)) \( \cup \) A' \( \setminus \) R',\( \mathcal{R}_j'\)),
flush (ys@[Ghost\(_sb\) A' L' R' W']) m, share (ys@[Ghost\(_sb\) A'
L' R' W']) \( S\))
(is (-,-,\( \_\)) \( \Rightarrow \) \( d^*\) (?ts-ys,?m-ys,?shared-ys))
by (auto simp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: i < length ?ts-ys
by auto

from i-bound' neq-i-j ts-i
have ts-ys-i: (?ts-ys!i = (\( \text{hd-prog } p_{sb} sb'\), Write True a (D, f) A L R W\# is', \( ?\theta\), ()),
\( D\),
acquired True ?take-sb O\(_sb\),release ?take-sb (dom \( S_{sb} \) \( \mathcal{R}_{sb} \))
by simp

note conflict-computation = steps-ys

642
from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
have safe: safe-delayed (\(ts-ys, m-ys, shared-ys\)).

with safe-delayedE [OF safe i-bound-ys ts-ys-i]
have a-unowned:

\(\forall j < \text{length } ts-ys. i \neq j \rightarrow (\text{let } (O_j) = \text{map owned } ts-ys[j \text{ in a } \notin O_j})\)
apply cases
apply (auto simp add: Let-def sb)
done
from a-A' a-unowned [rule-format, of j] neq-i-j j-bound leq A'R'
show False
by (auto simp add: Let-def)
qed

note a-notin-unforwarded-non-volatile-reads-drop = this

have valid-reads': valid-reads m_{sb'} ts_{sb'}
proof (unfold-locales)
  fix j p_j is_j O_j R_j D_j \theta_j sb_j
  assume j-bound: j < \text{length } ts_{sb'}
  assume ts-j: ts_{sb'} \{j = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j)\}
  show reads-consistent False O_j m_{sb'} sb_j
  proof (cases i=j)
    case True
    from reads-consist ts-j j-bound sb show \{thesis
      by (clarsimp simp add: True m_{sb'} Write_{sb} ts_{sb'} O_{sb'} volatile reads-consistent-pending-write-antimono)
    next
case False
    from j-bound have j-bound': j < \text{length } ts_{sb'}
      by (simp add: ts_{sb'}
    moreover from ts-j False have ts-j': ts_{sb'} \{j = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j)\}
      using j-bound by (simp add: ts_{sb'}
    ultimately have consis-m: reads-consistent False O_j m_{sb} sb_j
      by (rule valid-reads)
    from a-unowned-by-others [rule-format, OF j-bound' False] ts-j'
    have a-unowned:a \notin O_j \cup \text{all-acquired } sb_j
      by simp
    let ?take-sb_j = takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j
    let ?drop-sb_j = dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j
    from a-unowned acquired-reads-all-acquired [of True ?take-sb_j O_j]
    all-acquired-append [of ?take-sb_j ?drop-sb_j]
    have a-not-acq-reads: a \notin \text{acquired-reads } True ?take-sb_j O_j

643
by auto
moreover
note a-unfw= a-notin-unforwarded-non-volatile-reads-drop [OF j-bound' ts-j' False]
ultimately
show \( ? \)thesis
  using reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop [where
W=\{\} and
A=unforwarded-non-volatile-reads ?drop-sb \{\} \cup acquired-reads True ?take-sb \O_j and
m'= (m_{sb}(a:=v)), OF - - - consis-m]
  by (fastforce simp add: m_{sb}')
qed
qed

have valid-own': valid-ownership \( S_{sb}' \) ts_{sb}'
proof (intro-locale)
  show outstanding-non-volatile-refs-owned-or-read-only \( S_{sb}' \) ts_{sb}'
  proof
    fix j is \_ O_j R_j D_j \_ j sb \_ p_j
    assume j-bound: \( j \prec \) length ts_{sb}'
    assume ts_{sb}'-j= t_{sb}' j = (p_j,i\_j,\_ j sb \_ D_j,O_j,R_j)
    show non-volatile-owned-or-read-only False \( S_{sb}' \) O_j sb_j
    proof (cases j=i)
      case True
      from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb}-i]
      have non-volatile-owned-or-read-only False
       (\( S_{sb} \oplus W \) R \( \ominus A \) L) (\( O_{sb} \cup A - R \)) sb'
      by (auto simp add: sb Write_{sb} volatile non-volatile-owned-or-read-only-pending-write-antimono)
      then show \(?\)thesis
    using True i-bound ts_{sb}'-j
    by (auto simp add: ts_{sb}' S_{sb}' sb O_{sb}')
    next
      case False
      from j-bound have j-bound': \( j \prec \) length ts_{sb}
      by (auto simp add: ts_{sb}')
      with ts_{sb}'-j False i-bound
      have ts_{sb}-j= t_{sb} j = (p_j,i\_j,\_ j sb \_ D_j,O_j,R_j)
      by (auto simp add: ts_{sb}' )

    note nvo = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound' ts_{sb}-j]

    from read-only-unowned [OF i-bound ts_{sb}-i] R-owned
    have R \( \cap \) read-only \( S_{sb} = \{\} \)
    by auto
    with read-only-reads-unowned [OF j-bound' i-bound False ts_{sb}-j ts_{sb}-i] L-subset
    have \( \forall a \in \) read-only-reads
      (acquired True (takeWhile (Not \( o \) is-volatile-Write_{sb}) sb_j) \O_j)
      (dropWhile (Not \( o \) is-volatile-Write_{sb}) sb_j).
    a \( \in \) read-only \( S_{sb} \rightarrow a \in \) read-only (\( S_{sb} \oplus W \) R \( \ominus A \) L)
by (auto simp add: in-read-only-convs sb Write sb volatile)
  from non-volatile-owned-or-read-only-read-only-reads-eq' [OF nvo this]
  have non-volatile-owned-or-read-only False (\(S_{sb} \oplus R \ominus A\) \(O_j\) \(sb\)).
  thus ?thesis by (simp add: \(S_{sb}'\))
qed
qed
next
  show outstanding-volatile-writes-unowned-by-others \(ts_{sb}'\)
  proof (unfold-locales)
    fix \(i\) \(j\) \(p_1\) \(p_j\) \(is_1\) \(is_j\) \(xs_1\) \(xs_j\) \(sb_1\) \(sb_j\)
    assume \(i_1\)-bound: \(i_1 < \) length \(ts_{sb}'\)
    assume \(j\)-bound: \(j < \) length \(ts_{sb}'\)
    assume \(i_1\)-\(j\): \(i_1 \neq j\)
    assume \(ts-i_1\): \(ts_{sb}' \| i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)
    assume \(ts-j\): \(ts_{sb}' \| j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)\)
    show \((O_j \cup all-acquired sb_j) \cap outstanding-refs is-volatile-Write_{sb} sb_1 = \{\}\)
    proof (cases \(i = i\))
      case True
      from \(outstanding-volatile-writes-unowned-by-others [OF i_1\)-bound \(j\)-bound \(\prime\) \(neq-i-j\) \(ts_{sb} - i\) \(ts_{sb} - i'\)]\(ts_i\) False have \(ts_{i_1} = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)
      by (simp add: \(ts_{sb}'\) sb)
      show ?thesis
      using \(True\) \(i = i\) \(ts_{sb} - i\) \(ts_{sb} - i'\) i-bound
      next
      case False
      from \(j\)-bound have \(j\)-bound\(': j < \) length \(ts_{sb}\)
      by (simp add: \(ts_{sb}'\))
      from \(ts-j\) \(neq-i-j\) have \(ts-j_1\): \(ts_{sb}'_1\) = \((p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)\)
      by (simp add: \(ts_{sb}'\))
      from outstanding-volatile-writes-unowned-by-others [OF \(i\)-bound \(j\)-bound\(\prime\) \(neq-i-j\) \(ts_{sb} - i\) \(ts_{i_1} = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)]
      by (clarsimp simp add: \(S_{sb} \oplus R \ominus A\) \(O_j\) \(sb\)).
      then show ?thesis
      using \(True\) \(i = i\) \(ts_{sb} - i\) \(ts_{sb} - i'\) i-bound
      next
    case False
    from \(j\)-bound have \(j\)-bound\(': j < \) length \(ts_{sb}\)
    by (simp add: \(ts_{sb}'\))
    from \(ts-j\) False have \(ts-j_1\): \(ts_{sb}'_1\) = \((p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)\)
    by (simp add: \(ts_{sb}'\) sb)
    show ?thesis
    proof (cases \(j = i\))
      case True
      from \(outstanding-volatile-writes-unowned-by-others [OF \(i\)-bound \(j\)-bound\(\prime\) \(i\)- bound \(i\)-\(i\) \(ts_{i_1} = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)]\(ts_{sb} - i\) \(ts_{i_1} = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)
      have \((O_{sb} \cup all-acquired sb) \cap outstanding-refs is-volatile-Write_{sb} sb_1 = \{\}\).
      then show ?thesis
      using \(True\) \(i = i\) \(ts_{sb} - i\) \(ts_{sb} - i'\) i-bound
      by (auto simp add: sb Write sb volatile \(O_{sb}'\))
by (simp add: ts\_sb\’)
from outstanding-volatile-writes-unowned-by-others
[OF \(\text{i-bound}’ j-bound’ \text{i-bound}’ \text{ts-i’ ts-j’}\)]
show \((O_j \cup \text{all-acquired sb}) \cap \text{outstanding-refs is-volatile-Write}_{sb} \ sb_1 = \{\}\)
  qed
  qed
  qed
next
show read-only-reads-unowned ts\_sb’
proof
fix n m
fix p_n is_n \(\mathcal{O}_n R_n \emptyset_n sb_n\) p_m is_m \(\mathcal{O}_m R_m \emptyset_m sb_m\)
assume n-bound: \(n < \text{length ts}_{sb}\’\)
  and m-bound: \(m < \text{length ts}_{sb}\’\)
  and neq-n-m: \(n \neq m\)
  and nth: \(ts_{sb}\!’_n = (p_n, \text{is}_n, \emptyset_n, sb_n, D_n, O_n, R_n)\)
  and mth: \(ts_{sb}\!’_m = (p_m, \text{is}_m, \emptyset_m, sb_m, D_m, O_m, R_m)\)
from n-bound have n-bound’: \(n < \text{length ts}_{sb}\) by (simp add: ts\_sb\’)
from m-bound have m-bound’: \(m < \text{length ts}_{sb}\) by (simp add: ts\_sb\’)
show \((O_m \cup \text{all-acquired sb}) \cap \text{read-only-reads (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n))} \ (\text{dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n}) = \{\}\)
proof (cases \(m=i\))
  case True
  with neq-n-m have neq-n-i: \(n \neq i\)
  by auto
  with n-bound nth i-bound have nth’: \(ts_{sb}\!’_n = (p_n, \text{is}_n, \emptyset_n, sb_n, D_n, O_n, R_n)\)
  by (auto simp add: ts\_sb\’)
  note read-only-reads-unowned [OF \(n-bound’ i-bound\) neq-n-i nth’ ts\_sb-i]
  then
  show ?thesis
  using True ts\_sb-i neq-n-i nth m-bound’ m-bound’ L-subset
  by (auto simp add: ts\_sb’ O\_sb’ sb Write\_sb volatile)
next
  case False
  note neq-m-i = this
  with m-bound mth i-bound have mth’: \(ts_{sb}\!’_m = (p_m, \text{is}_m, \emptyset_m, sb_m, D_m, O_m, R_m)\)
  by (auto simp add: ts\_sb\’)
  show ?thesis
  proof (cases \(n=i\))
  case True
  from read-only-reads-append [of \(O_{sb} \cup A - R\)]
  (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n)
  (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n)]
  have read-only-reads
  (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n) \(O_{sb} \cup A - R\))
  (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_n) \(\subseteq \text{read-only-reads (O}_{sb} \cup A - R\) sb_n

646
by auto

with \texttt{ts}_{sb}-i \texttt{nth mth neq-i m-bound’ True}
read-only-reads-unowned \([\texttt{OF i-bound m-bound’ False [symmetric]} \texttt{ts}_sb-i \texttt{mth}]\)

\texttt{show} \ ?thesis
  by (auto simp add: \texttt{ts}_sb′ \texttt{sb} \texttt{O}_{sb}′ \texttt{Write}_{sb} \texttt{volatile})
next
case False
with \texttt{n-bound nth i-bound have nth′: ts}_sb\texttt{n} = (p_n, is_n, \emptyset_n, sb_n, D_n, O_n, R_n)
by (auto simp add: ts_{sb}′)

from read-only-reads-unowned \([\texttt{OF n-bound’ m-bound’ neq-n-m nth’mth’]} \texttt{False neq-m-i}\)
\texttt{show} \ ?thesis
by (clarsimp)
qed

qed

next

\texttt{show} ownership-distinct \texttt{ts}_sb′
proof (unfold-locales)
fix i_1 \ j \ p_1 \ is_1 \ \texttt{O}_1 \ \texttt{R}_1 \ D_1 \ xs_1 \ sb_1 \ p_j \ is_j \ \texttt{O}_j \ \texttt{R}_j \ D_j \ xs_j \ sb_j
assume \texttt{i1-bound: i} \_1 < \texttt{length ts}_sb′
assume \texttt{j-bound: j} < \texttt{length ts}_sb′
assume \texttt{i1-j: i} \_1 \# \ j
assume \texttt{ts-i1: ts}_sb′!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)
assume \texttt{ts-j: ts}_sb′!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
\texttt{show} (\texttt{O}_1 \cup \texttt{all-acquired sb}_1) \cap (\texttt{O}_j \cup \texttt{all-acquired sb}_j) = \{\}
proof (cases \texttt{i1=i})
case True
with \texttt{i1-j have i-j: i} \# \ j
by simp

from \texttt{j-bound have j-bound′: j} < \texttt{length ts}_sb
by (simp add: \texttt{ts}_sb′)
hence \texttt{j-bound′: j} < \texttt{length (map owned ts}_sb)
by simp
from \texttt{ts-j i-j have ts-j′: ts}_sb′!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
by (simp add: \texttt{ts}_sb′)

from ownership-distinct \([\texttt{OF i-bound j-bound′ i-j ts}_sb-i \texttt{ts-j′}]
\texttt{show} \ ?thesis
using \texttt{ts}_sb-i \texttt{True ts-i1 i-bound \texttt{O}_{sb}′}
by (auto simp add: \texttt{ts}_sb′!sb \texttt{Write}_{sb} \texttt{volatile})
next
case False
note \texttt{i1-i = this}
from \texttt{i1-bound have i1-bound′: i} \_1 < \texttt{length ts}_sb
by (simp add: \texttt{ts}_sb′)
hence \texttt{i1-bound′: i} \_1 < \texttt{length (map owned ts}_sb)
by simp
from \texttt{ts-i1 False have ts-i1′: ts}_sb′!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)
647
by (simp add: \ts_{\text{sb}}')

show \text{thesis}

do cases (j = i)

case True

from ownership-distinct [OF \i_1 - bound' \i-bound \i_1 - \ts_{\text{sb}} - \i-bound \i_1 - \ts - \i-bound \i_1 - \ts - \i-bound]

show \text{thesis}

using \ts_{\text{sb}} - \i-bound True \ts - j \ i-bound \O

by (auto simp add: \ts_{\text{sb}}' \sb \Write_{\text{sb}} \text{volatile})

next

case False

from \j-bound have \j-bound': \j < \length \ts_{\text{sb}}

by (simp add: \ts_{\text{sb}}')

from \ts - j False have \ts - j': \ts_{\text{sb}}! \j = (\p_{\j}, \i_{\j}, \xs_{\j}, \sb_{\j}, \D_{\j}, \O_{\j}, \R_{\j})

by (simp add: \ts_{\text{sb}}')

from ownership-distinct [OF \i_1 - bound' \j-bound' \i_1 - \ts - \j-bound \i_1 - \ts - \j-bound]

show ?thesis.

qed

qed

qed

have valid-sharing': valid-sharing (\S_{\text{sb}} \oplus \W \R \ominus \A \L) \ts_{\text{sb}}'

proof (intro-locales)

show outstanding-non-volatile-writes-unshared (\S_{\text{sb}} \oplus \W \R \ominus \A \L) \ts_{\text{sb}}'

proof (unfold-locales)

fix \j \p_{\j} \i_{\j} \O_{\j} \D_{\j} \acq_{\j} \xs_{\j} \sb_{\j}

assume \j-bound: \j < \length \ts_{\text{sb}}'

assume \jth: \ts_{\text{sb}}'! \j = (\p_{\j}, \i_{\j}, \xs_{\j}, \sb_{\j}, \D_{\j}, \O_{\j}, \R_{\j})

show non-volatile-writes-unshared (\S_{\text{sb}} \oplus \W \R \ominus \A \L) \sb_{\j}

proof (cases i = \j)

  case True

  with outstanding-non-volatile-writes-unshared [OF \i-bound \ts_{\text{sb}} - \i-bound]

  \i-bound \jth in \ts_{\text{sb}} - i show ?thesis

  by (clarsimp simp add: \ts_{\text{sb}}' \sb \Write_{\text{sb}} \text{volatile})

next

  case False

  from \j-bound have \j-bound': \j < \length \ts_{\text{sb}}

  by (auto simp add: \ts_{\text{sb}}')

  from \jth False have \jth': \ts_{\text{sb}}'! \j = (\p_{\j}, \i_{\j}, \xs_{\j}, \sb_{\j}, \D_{\j}, \O_{\j}, \R_{\j})

  by (auto simp add: \ts_{\text{sb}}')

  from outstanding-non-volatile-writes-unshared [OF \i-bound' \jth']

  have unshared: non-volatile-writes-unshared \S_{\text{sb}} \sb_{\j}.

  have \forall \a \in \text{dom} (\S_{\text{sb}} \oplus \W \R \ominus \A \L) \setminus \text{dom} \S_{\text{sb}}. \a \notin \text{outstanding-refs} \is-non-volatile-Write_{\text{sb}} \sb_{\j}

  proof

  { fix \a
    assume \a-in: \a \in \text{dom} (\S_{\text{sb}} \oplus \W \R \ominus \A \L) \setminus \text{dom} \S_{\text{sb}}

    hence \a-R: \a \in \R

    648
by clarsimp
assume a-in-j: a ∈ outstanding.refs is-non-volatile.Write\textsubscript{sb} \textsubscript{j}

have False

proof –
from non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF outstanding-non-volatile-refs-owned-or-read-only [OF j-bound' jth']]
a-in-j
have a ∈ O\textsubscript{j} ∪ all-acquired sb\textsubscript{j}
by auto

moreover
with ownership-distinct [OF i-bound j-bound' False ts\textsubscript{sb}-i jth' a-R R-owned]
show False
by blast
qed

} thus ?thesis by blast
qed

from non-volatile-writes-unshared-no-outstanding-non-volatile-Write\textsubscript{sb}
[OF unshared this]
show ?thesis .
qed
qed
next

show sharing-consis (S\textsubscript{sb} ⊕ W R ⊆ A L) ts\textsubscript{sb}'
proof (unfold-locales)

fix j p\textsubscript{j} is\textsubscript{j} O\textsubscript{j} R\textsubscript{j} D\textsubscript{j} xs\textsubscript{j} sb\textsubscript{j}
assume j-bound: j < length ts\textsubscript{sb}'
assume jth: ts\textsubscript{sb}' ! j = (p\textsubscript{j},is\textsubscript{j},xs\textsubscript{j},sb\textsubscript{j},D\textsubscript{j},O\textsubscript{j},R\textsubscript{j})
show sharing-consistent (S\textsubscript{sb} ⊕ W R ⊆ A L) O\textsubscript{j} sb\textsubscript{j}
proof (cases i=j)

  case True
  with i-bound jth ts\textsubscript{sb}-i sharing-consis [OF i-bound ts\textsubscript{sb}-i]
  show ?thesis
  by (clarsimp simp add: ts\textsubscript{sb}' sb Write\textsubscript{sb} volatile O\textsubscript{sb}')

next

  case False
  from j-bound have j-bound': j < length ts\textsubscript{sb}
  by (auto simp add: ts\textsubscript{sb}')

  from jth False have jth': ts\textsubscript{sb}' ! j = (p\textsubscript{j},is\textsubscript{j},xs\textsubscript{j},sb\textsubscript{j},D\textsubscript{j},O\textsubscript{j},R\textsubscript{j})
  by (auto simp add: ts\textsubscript{sb}')

  from sharing-consis [OF j-bound' jth']
  have consis: sharing-consistent S\textsubscript{sb} O\textsubscript{j} sb\textsubscript{j}.

  have acq-cond: all-acquired sb\textsubscript{j} ∩ dom S\textsubscript{sb} − dom (S\textsubscript{sb} ⊕ W R ⊆ A L) = {}
  proof –
  {

  649
fix a
assume a-acq: a ∈ all-acquired sb
assume a ∈ dom S
assume a-L: a ∈ L
have False
proof —
from ownership-distinct [OF i-bound j-bound′ False ts\sb′-i jth]
have A ∩ all-acquired sb = {}
  by (auto simp add: sb Write\sb volatile)
with a-acq a-L L-subset
show False
  by blast
qed
}
thus ?thesis
by auto
qed

{\}
fix a
assume a-uns: a ∈ all-unshared sb
assume a /∈ L
assume a-R: a ∈ R
have False
proof —
from unshared-acquired-or-owned [OF consis] a-uns
have a ∈ all-acquired sb ∪ O
  by auto
with ownership-distinct [OF i-bound j-bound′ False ts\sb′-i jth′] R-owned a-R
show False
  by blast
qed

next
show read-only-unowned (S + W R ⊕ A) ts\sb′
proof
fix j p\j is\j O\j R\j D\j xs\j sb\j
assume j-bound: j < length ts\sb
assume jth: ts\sb′! j = (p\j, is\j, xs\j, sb\j, D\j, O\j, R\j)
show O\j ∩ read-only (S + W R ⊕ A) = {}
proof (cases i=j)
case True
from read-only-unowned [OF i-bound ts_{sb}^{-i}] R-owned A-R
have \((O_{sb} \cup A - R) \cap \text{read-only} (S_{sb} \oplus_w R \ominus_A L) = \{\}\)
by (auto simp add: in-read-only-convs)
   with jth ts_{sb}^{-i} i-bound True
   show \(?thesis
by (auto simp add: O_{sb} \prime ts_{sb} \prime)
next
case False
from j-bound have j-bound': j < length ts_{sb}
by (auto simp add: ts_{sb} \prime)
   with False jth have jth': ts_{sb} ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
by (auto simp add: ts_{sb} \prime)
from read-only-unowned [OF j-bound' jth'] R-owned
have \((O_{sb} \cup A) \cap O_j = \{\}\)
by (auto simp add: sb Write \sb volatile)
   moreover note R-owned A-R
   ultimately show \(?thesis
by (fastforce simp add: in-read-only-convs split: if-split-asm)
qed
qed
next
show unowned-shared \((S_{sb} \oplus_w R \ominus_A L) \cap \text{ts}_{sb} \prime\)
proof (unfold-locales)
   show \(- \bigcup ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). O) \cdot \text{set ts}_{sb} \prime) \subseteq \text{dom} (S_{sb} \oplus_w R \ominus_A L)\)
   proof -
   have s: \(\bigcup ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). O) \cdot \text{set ts}_{sb} \prime) = \)
   \(\bigcup ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). O) \cdot \text{set ts}_{sb}) \cup A - R\)
apply (unfold ts_{sb} \prime O_{sb} \prime)
apply (rule acquire-release-ownership-nth-update [OF R-owned i-bound ts_{sb}^{-i}])
apply (rule local.ownership-distinct-axioms)
done
   note unowned-shared L-subset A-R
   then
   show \(?thesis
apply (simp only: s)
apply auto
done
qed
qed
next
show no-outstanding-write-to-read-only-memory \((S_{sb} \oplus_w R \ominus_A L) \cap \text{ts}_{sb} \prime\)
proof
fix \(j, p_j, \text{is}_j, O_j, R_j, D_j, \text{acq}_j, \text{xs}_j, \text{sb}_j\)
assume j-bound: j < length ts_{sb}'
assume jth: ts_{sb}' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
show no-write-to-read-only-memory (S_{sb} \oplus_W R \ominus_A L) sb_j

proof (cases i=j)
case True
  with jth ts_{sb}-i i-bound no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb}-i]
  show ?thesis
  by (auto simp add: sb ts sb' Write volatile)

next
  case False
  from j-bound have j-bound': j < length ts_{sb}
  by (auto simp add: ts_{sb}')
  with False jth have jth': ts_{sb} ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
  by (auto simp add: ts sb' Write volatile)

  note dist = ownership-distinct [OF i-bound j-bound' False ts_{sb}-i jth']
  from non-volatile-owned-or-read-only-non-volatile-writes
  [OF outstanding-non-volatile-refs-owned-or-read-only [OF j-bound' jth']] dist
  have outstanding-refs is-non-volatile-Write sb_{sb} sb_j \cap O_{sb} = {}
    by auto
  moreover
  from outstanding-volatile-writes-unowned-by-others [OF j-bound' i-bound]
    False [symmetric] jth' ts_{sb}-i ]
  have outstanding-refs is-volatile-Write sb_{sb} sb_j \cap O_{sb} = {}
    by auto
  ultimately have outstanding-refs is-Write sb_{sb} sb_j \cap O_{sb} = {}
    by (auto simp add: misc-outstanding-refs-convs)
  with R-owned
  show ?thesis by blast
  qed
  then
    have \forall a \in outstanding-refs is-Write sb_{sb} sb_j.
a \in read-only (S_{sb} \oplus_W R \ominus_A L) \rightarrow a \in read-only S_{sb}
    by (auto simp add: in-read-only-convs)
  
    from no-write-to-read-only-memory-read-only-reads-eq [OF nw this]
    show ?thesis .
  qed
  qed

qed

from direct-memop-step.WriteVolatile [OF]
have (Write True a (D, f) A L R W# is',
  ?\theta, (), m, D, acquired True ?take-sb O_{sb}, release ?take-sb (dom S_{sb}) R_{sb}, S) \rightarrow
  (is', ?\theta, (), m (a := v), True, acquired True ?take-sb O_{sb} \cup A - R, Map.empty, S
  \oplus_W R \ominus_A L)
by (simp add: f-v' [symmetric])

from direct-computation.Memop [OF i-bound' ts-i this]
have store-step:
  \((ts, m, \mathcal{S}) \Rightarrow_d \langle \?ts', m(a := v), \mathcal{S} \oplus_w R \oplus_A L \rangle\).

have sb'-split:
  \(sb' = \text{takeWhile} (\neg \circ \text{is-volatile-Write}_{sb}) \; sb' \circ @ \text{dropWhile} (\neg \circ \text{is-volatile-Write}_{sb}) \; sb'\)
  by simp

from reads-consis
have no-vol-reads: outstanding-refs is-volatile-Read_{sb} sb' = {} 
  by (simp add: sb Write_{sb} True)

hence outstanding-refs is-volatile-Read_{sb} \text{(takeWhile} (\neg \circ \text{is-volatile-Write}_{sb}) \; sb'\) = {}
  by (auto simp add: outstanding-refs-conv dest: set-takeWhileD)

moreover
have outstanding-refs is-volatile-Write_{sb} \text{(takeWhile} (\neg \circ \text{is-volatile-Write}_{sb}) \; sb'\) = {}

proof
  have \(\forall r \in \text{set} \text{(takeWhile} (\neg \circ \text{is-volatile-Write}_{sb}) \; sb') \neg \text{(is-volatile-Write}_{sb} \; r\)
    by (auto dest: set-takeWhileD)

  thus \(?\text{thesis}\)
    by (simp add: outstanding-refs-conv)
  qed

ultimately
have no-volatile:
  outstanding-refs is-volatile (takeWhile (\neg \circ \text{is-volatile-Write}_{sb}) \; sb') = {}
  by (auto simp add: outstanding-refs-conv is-volatile-split)

moreover

from no-vol-reads have \(\forall r \in \text{set} \; sb', \neg \text{is-volatile-Read}_{sb} \; r\)
  by (fastforce simp add: outstanding-refs-conv is-volatile-Read_{sb} def split: memref.splits)

hence \(\forall r \in \text{set} \; sb', (\neg \circ \text{is-volatile-Write}_{sb}) \; r = (\neg \circ \text{is-volatile}) \; r\)
  by (auto simp add: is-volatile-split)

hence takeWhile-eq: \text{(takeWhile} (\neg \circ \text{is-volatile-Write}_{sb}) \; sb') = \text{(takeWhile} (\neg \circ \text{is-volatile}) \; sb')

  apply --

  apply (rule takeWhile-cong)

  apply auto

  done

from leq

have leq': \text{length} \; ts_{sb} = \text{length} \; \?ts'
  by simp

hence i-bound-ts': \text{i < length} \; \?ts' \text{ using} i-bound by simp
from is'\text{-}sim
have is'\text{-}sim-split:
  instrs
  (takeWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime @
dropWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime) @ is_{sb} =
is' @ prog-instrs (takeWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime @
dropWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime)
by (simp add: sb'\text{-}split [symmetric])

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing \(S_{sb}\)
ts_{sb})
i-bound ts_{sb}\text{-}i reads-consis]
have reads-consistent True (acquired True \(?\text{take-sb} O_{sb}\) m (Write_{sb} True a (D,f) v A L
R W\#sb \prime)
by (simp add: m sb Write_{sb} volatile)

hence reads-consistent True (acquired True \(?\text{take-sb} O_{sb} \cup A - R\) (m(a:=v)) sb \prime
by simp
from reads-consistent-takeWhile [OF this]
have r-consis' : reads-consistent True (acquired True \(?\text{take-sb} O_{sb} \cup A - R\) (m(a:=v))
  (takeWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime).

from last-prog have last-prog-sb \prime: last-prog p_{sb} sb \prime = p_{sb}
by (simp add: sb Write_{sb} )

from valid-write-sops [OF i-bound ts_{sb}\text{-}i]
have \(\forall\text{sop} \in \text{write-sops sb} \prime. \text{valid-sop sop}\)
  by (auto simp add: sb Write_{sb})

hence valid-sop' : \(\forall\text{sop} \in \text{write-sops} \text{(takeWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime)}.\)
  valid-sop sop
  by (fastforce dest: set-takeWhileD simp add: in-write-sops-conv)

from no-volatile
have no-volatile-Read_{sb}:
  outstanding-refs is-volatile-Read_{sb} (takeWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime) =
  \{
  \}
  by (auto simp add: outstanding-refs-conv is-volatile-Read_{sb}\text{-def split: memref.splits})
from flush-store-buffer-append [OF i-bound-ts' is'\text{-}sim-split, simplified,\nOF causal-program-history-sb' ts'\text{-}i refl last-prog-sb' r-consis' hist-consis'\nvalid-sop' dist-sb' no-volatile-Read_{sb}\text{-volatile-reads-consistent} [OF no-volatile-Read_{sb}],
where \(S=(S \oplus W R \ominus A L)\)

obtain is'' where
is''\text{-}sim: instrs (dropWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime) @ is_{sb} =
is'' @ prog-instrs (dropWhile (Not \circ\text{is-volatile-Write}_{sb}) sb \prime) and

654
steps: (?ts', m(a := v), S ⊕ W R ⊆ A L) ⇒ d^∗
(ts| i := (last-prog (hd-prog p#sb (dropWhile (Not o is-volatile-Write sb') sb'))
(takeWhile (Not o is-volatile-Write sb) sb'),
is'’;
θsb |' (dom θsb −
read-tmps (dropWhile (Not o is-volatile-Write sb') sb')),
(), True, acquired True (takeWhile (Not o is-volatile-Write sb) sb')
(acquired True ?take-sb \{sb\} ∪ A − R),
release (takeWhile (Not o is-volatile-Write sb) sb')
(dom (S ⊕ W R ⊆ A L)) Map.empty),
flush (takeWhile (Not o is-volatile-Write sb) sb') (m(a := v)),
share (takeWhile (Not o is-volatile-Write sb) sb') (S ⊕ W R ⊆ A L)

by (auto)

note sim-flush = r-rtranclp-rtranclp [OF store-step steps]

moreover
note flush-commute =
flush-flush-all-until-volatile-write-Write sb-volatile-commute [OF i-bound ts#sb-i [simplified sb Write sb True]
outstanding-refs-is-Write sb-takeWhile-disj a-notin-others']

from last-prog-hd-prog-append' [where sb=(takeWhile (Not o is-volatile-Write sb) sb')
and sb'==(dropWhile (Not o is-volatile-Write sb) sb'),
simplified sb'=split [symmetric], OF hist-consis' last-prog-sb']
have last-prog-eq:
last-prog (hd-prog p#sb (dropWhile (Not o is-volatile-Write sb) sb'))
(takeWhile (Not o is-volatile-Write sb) sb') =
hd-prog p#sb (dropWhile (Not o is-volatile-Write sb) sb').

have take-empty: takeWhile (Not o is-volatile-Write sb) (r#sb) = []
by (simp add: Write sb True)

have dist-sb': ∀ i p is O R D \ ø sb.
i < length ts#sb −−>
ts#sb ! i = (p, is, \ ø, sb, D, O, R) −−>
(all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∪
all-unshared (takeWhile (Not o is-volatile-Write sb) sb) ∪
all-acquired (takeWhile (Not o is-volatile-Write sb) sb) ∩
(all-shared (takeWhile (Not o is-volatile-Write sb) sb') ∪
all-unshared (takeWhile (Not o is-volatile-Write sb) sb') ∪
all-acquired (takeWhile (Not o is-volatile-Write sb) sb')) =
{}

proof −
{fix j p j is j O j R j D j \ ø j sb j x
assume \( j \)-bound: \( j < \text{length} \ ts_{sb} \)
assume \( j \text{th}: \ ts_{sb}[j] = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) \)
assume \( x\)-shared: \( x \in \text{all-shared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}) sb_j) \cup \text{all-unshared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}) sb_j) \cup \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}) sb_j) \)
assume \( x\)-sb\': \( x \in (\text{all-shared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}) sb') \cup \text{all-unshared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}) sb') \cup \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}) sb') \)

have False
proof (cases \( i=j \))
case True with \( x\)-shared \( ts_{sb}-i \) \( j \text{th} \)
show False by (simp add: sb volatile Write)
next
case False
from \( x\)-shared all-shared-acquired-or-owned [OF sharing-consis [OF \( j \)-bound \( j \text{th} \)]
unshared-acquired-or-owned [OF sharing-consis [OF \( j \)-bound \( j \text{th} \)]
all-shared-append [of (takeWhile (\text{Not} \circ \text{is-volatile-Write}) sb_j)]
(dropWhile (\text{Not} \circ \text{is-volatile-Write} \ ts_{sb}) sb_j)
all-unshared-append [of (takeWhile (\text{Not} \circ \text{is-volatile-Write}) sb_j)]
(dropWhile (\text{Not} \circ \text{is-volatile-Write} \ ts_{sb}) sb_j)
all-acquired-append [of (takeWhile (\text{Not} \circ \text{is-volatile-Write}) sb_j)]
(dropWhile (\text{Not} \circ \text{is-volatile-Write} \ ts_{sb}) sb_j)

have \( x \in \text{all-acquired} sb_j \cup O_j \)
by auto
moreover
from \( x\)-sb\' all-shared-acquired-or-owned [OF sharing-consis [OF \( i \)-bound \( ts_{sb}-i \)]
unshared-acquired-or-owned [OF sharing-consis [OF \( i \)-bound \( ts_{sb}-i \)]
all-shared-append [of (takeWhile (\text{Not} \circ \text{is-volatile-Write}) sb')]\)
(dropWhile (\text{Not} \circ \text{is-volatile-Write} \ ts_{sb}) sb')\)
all-unshared-append [of (takeWhile (\text{Not} \circ \text{is-volatile-Write}) sb')]\)
(dropWhile (\text{Not} \circ \text{is-volatile-Write} \ ts_{sb}) sb')\)
all-acquired-append [of (takeWhile (\text{Not} \circ \text{is-volatile-Write}) sb')]\)
(dropWhile (\text{Not} \circ \text{is-volatile-Write} \ ts_{sb}) sb')\)

have \( x \in \text{all-acquired} sb \cup O_{sb} \)
by (auto simp add: sb Write volatile)
moreover
note ownership-distinct [OF \( i \)-bound \( j \)-bound False \( ts_{sb}-i \) \( j \text{th} \)]
ultimately show False by blast
qed

thus \( ?\text{thesis} \) by blast
qed

have dist-R-L-A: \( \forall j \ p \ is \ O \ R \ D \ \emptyset \ sb. \)
\( j < \text{length} \ ts_{sb} \longrightarrow i \neq j \longrightarrow \)
\( ts_{sb} ! j = (p, is, \emptyset, sb, D, O, R) \longrightarrow \)
\( (\text{all-shared} sb \cup \text{all-unshared} sb \cup \text{all-acquired} sb) \cap (R \cup L \cup A) = \{\} \)
proof -
{
fix \( j \ p_j \ is_j \ O_j \ R_j \ D_j \ \emptyset \ sb_j \ x \)
assume \( j \)-bound: \( j < \) length \( ts_{sb} \)
assume neq-i-j: \( i \neq j \)
assume jth: \( ts_{sb}[j] = (p_i, is_j, \theta_j, D_j, O_j, R_j) \)
assume x-shared: \( x \in \) all-shared \( sb_j \cup \) all-unshared \( sb_j \cup \) all-acquired \( sb_j \)
assume x-R-L-A: \( x \in R \cup L \cup A \)

\begin{align*}
& \text{have False} \\
& \text{proof} - \\
& \quad \text{from x-shared all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]} \\
& \quad \quad \quad \text{unshared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]} \\
& \quad \quad \quad \text{have} \ x \in \text{all-acquired} \ sb_j \cup O_j \\
& \quad \quad \quad \quad \text{by auto} \\
& \quad \quad \quad \text{moreover} \\
& \quad \quad \quad \text{from x-R-L-A R-owned L-subset} \\
& \quad \quad \quad \quad \text{have} \ x \in \text{all-acquired} \ sb \cup O_{sb} \\
& \quad \quad \quad \quad \quad \text{by (auto simp add: sb Write}_{sb} \text{ volatile)} \\
& \quad \quad \quad \text{moreover} \\
& \quad \quad \quad \text{note ownership-distinct [OF i-bound j-bound neq-i-j ts}_{sb}-i jth] \\
& \quad \quad \quad \text{ultimately show False by blast} \\
& \quad \text{qed} \\
& \text{thus} \ \text{thesis by blast} \\
& \text{qed} \\
& \text{from local.ownership-distinct-axioms have ownership-distinct ts}_{sb}. \\
& \text{from local.sharing-consis-axioms have sharing-consis } S_{sb} \ \text{ts}_{sb}. \\
& \text{note share-commute=} \\
& \quad \text{share-all-until-volatile-write-flush-commute [OF take-empty (ownership-distinct ts}_{sb}) \ (sharing-consis } S_{sb} \ \text{ts}_{sb}) \ \text{i-bound ts}_{sb}-i \ \text{dist-sb} \ \text{dist-R-L-A]} \\
& \text{have rel-commute-empty:} \\
& \quad \text{release (takeWhile (Not \circ is-volatile-Write}_{sb}) \ sb') \ (dom } S \cup R - L) \ \text{Map.empty} = \\
& \quad \quad \text{release (takeWhile (Not \circ is-volatile-Write}_{sb}) \ sb') \ (dom } S_{sb} \cup R - L) \\
& \text{Map.empty} \\
& \text{proof} - \\
& \quad \{ \\
& \quad \quad \text{fix a} \\
& \quad \quad \text{assume a-in: } a \in \text{all-shared} \ (\text{takeWhile (Not } \circ \text{is-volatile-Write}_{sb}) \ sb') \\
& \quad \quad \text{have } (a \in (\text{dom } S \cup R - L)) = (a \in (\text{dom } S_{sb} \cup R - L)) \\
& \quad \quad \text{proof} - \\
& \quad \quad \text{from all-shared-acquired-or-owned [OF sharing-consis [OF i-bound ts}_{sb}-i]] a-in} \\
& \quad \quad \text{all-shared-append [of (takeWhile (Not } \circ \text{is-volatile-Write}_{sb}) \ sb') \ (\text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb}) \ sb')] \\
& \quad \quad \text{have } a \in O_{sb} \cup \text{all-acquired } sb \\
& \quad \quad \quad \text{by (auto simp add: sb Write}_{sb} \text{ volatile)} \\
& \quad \quad \quad \text{from share-all-until-volatile-write-thread-local [OF (ownership-distinct ts}_{sb}) \ (sharing-consis } S_{sb} \ \text{ts}_{sb}) \ \text{i-bound ts}_{sb}-i \ \text{this]} \\
\end{align*}
have $S$ $a = S_{sb}$ $a$
  by (auto simp add: sb Write$_{sb}$ volatile $S$)
then show ?thesis
  by (auto simp add: domIf)
qed

} then show ?thesis
  apply -
  apply (rule release-all-shared-exchange)
  apply auto
  done
qed

{ fix $j$ $p_j$ is$_j$ $O_j$ $R_j$ $\emptyset_j$ $sb_j$ $x$
  assume jth: $ts_{sb}[j] = (p_j, is_j, \emptyset_j, sb_j, D_j, O_j, R_j)$
  assume j-bound: $j < \text{length} ts_{sb}$
  assume neq: $i \neq j$
  have release (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$)
    (dom $S_{sb} \cup R - L$) $R_j$
  = release (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$)
    (dom $S_{sb}$) $R_j$
proof -
  { fix $a$
    assume a-in: $a \in \text{all-shared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j)$
    have $(a \in (\text{dom} S_{sb} \cup R - L)) = (a \in \text{dom} S_{sb})$
    proof -
      from ownership-distinct [OF i-bound j-bound neq ts$_{sb}$-i jth]
      have $A - \cap (O_j \cup \text{all-acquired} sb_j) = \{\}$
      by (auto simp add: sb Write$_{sb}$ volatile)
    from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]] a-in
    all-shared-append [of (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$)
    (dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$)]
    have a-in: $a \in O_j \cup \text{all-acquired sb}_j$
    by auto
    with ownership-distinct [OF i-bound j-bound neq ts$_{sb}$-i jth]
    have $a \notin (C_{ab} \cup \text{all-acquired sb})$ by auto
    with A-dist R-owned A-R A-shared-owned L-subset a-in
    obtain $a \notin R$ and $a \notin L$
    by fastforce
    then show ?thesis by auto
  qed
  }
then
show ?thesis

658
apply –
apply (rule release-all-shared-exchange)
apply auto
done
qed

}

note release-commute = this

have (ts \(sb\) [\(i := (p_{sb},i\_{sb},sb',D_{sb},O_{sb} \cup A - R,\text{Map}.\text{empty})\],m_{sb}(a:=v),S_{sb}')] \sim
(ts[i := (\text{last-prog}\ (\text{hd-prog}\ p_{sb})\ (\text{dropWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb'))
\ (\text{takeWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb')),
is''',
\(\theta_{sb}'\)\ |\ (\text{dom}\ \(\theta_{sb}'\) –
\text{read-tmps}\ (\text{dropWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb')),
(),\text{True},\text{acquired} \text{True}\ (\text{takeWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb')
\ (\text{acquired} \text{True} ?\text{take-sb}\ O_{sb} \cup A - R),
\text{release}\ (\text{takeWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb')
\ (\text{dom}\ (S \oplus W R \ominus A L))\ \text{Map}.\text{empty}),
\text{flush}\ (\text{takeWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb')\ (m(a := v)),
\text{share}\ (\text{takeWhile}\ (\text{Not} \circ \text{is-volatile-Write}_{sb})\ sb')\ (S \oplus W R \ominus A L))
apply (rule sim-configintros)
apply (simp add: flush-commute m)
apply (clarsimp simp add: \(S_{sb}'\) \(S\) share-commute simp del: restrict-restrict)
using leq
apply simp
using i-bound i-bound' ts-sim \(D\)
apply (clarsimp simp add: Let-def nth-list-update is''-sim last-prog-eq sb Write_{sb} volatile \(S_{sb}'\)
rel-commute-empty
split: if-split-asn )
apply (rule conjI)
apply blast
apply clarsimp
apply (frule (2) release-commute)
apply clarsimp
apply fastforce
done

ultimately

show \(?\text{thesis}\)

using valid-own' valid-hist' valid-reads' valid-sharing' \text{tmss-distinct}'
valid-dd' valid-sops' load-tmps-fresh' enough-flushs'
valid-program-history' valid'
m_{sb}' S_{sb}' ts_{sb}'

by (auto simp del: fun-upd-apply simp add: \(O_{sb}'\ \mathcal{R}_{sb}'\) )

next

case False

note non-vol = this
from flush Write\textsubscript{sb} False  

obtain  
\begin{align*}
O_{sb}' & : O_{sb} = O_{sb} & \text{and} \\
S_{sb}' & : S_{sb} = S_{sb} & \text{and} \\
R_{sb}' & : R_{sb} = R_{sb}
\end{align*}

by cases (auto simp add: sb)  

from non-volatile-owned non-vol have a-owned: a \in O_{sb}  
by simp  

\{  
fix j  
fix p\_j is\_sb j D\_sb j \_j R\_sb j
assume j-bound: j < length ts\_sb
assume ts\_sb-j: ts\_sb[j]= (p\_j, is\_sb j, D\_sb j, O\_j, R\_j)
assume neq-i-j: i \neq j
have a \notin unforwarded-non-volatile-reads (dropWhile (Not \circ is-volatile-Write\textsubscript{sb}) sb\_j) \{} 
proof 
let ?take-sb\_j = takeWhile (Not \circ is-volatile-Write\textsubscript{sb}) sb\_j
let ?drop-sb\_j = dropWhile (Not \circ is-volatile-Write\textsubscript{sb}) sb\_j
assume a-in: a \in unforwarded-non-volatile-reads ?drop-sb\_j \{}

from a-unowned-by-others [rule-format, OF j-bound neq-i-j] ts\_sb-j
obtain a-unowned: a \notin O\_j \text{ and} a-unacq: a \notin all-acquired sb\_j  
by auto  

with all-acquired-append [of ?take-sb\_j ?drop-sb\_j]
acquired-takeWhile-non-volatile-Write\textsubscript{sb} [of sb\_j O\_j]
have a-unacq-take: a \notin acquired True ?take-sb\_j O\_j
by (auto )  

note nvo-j = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound ts\_sb-j]  

from non-volatile-owned-or-read-only-drop [OF nvo-j]  
have nvo-drop-j: non-volatile-owned-or-read-only True (share ?take-sb\_j S\_sb)  
(acquired True ?take-sb\_j O\_j) ?drop-sb\_j
from in-unforwarded-non-volatile-reads-non-volatile-Read\textsubscript{sb} [OF a-in]
have a-in': a \in outstanding-refs is-non-volatile-Read\textsubscript{sb} ?drop-sb\_j.

from non-volatile-owned-or-read-only-outstanding-refs [OF nvo-drop-j] a-in'
have a \in acquired True ?take-sb\_j O\_j \cup all-acquired ?drop-sb\_j \cup
read-only-reads (acquired True ?take-sb\_j O\_j) ?drop-sb\_j
by (auto simp add: misc-outstanding-refs-convs)

moreover  
from acquired-append [of True ?take-sb\_j ?drop-sb\_j O\_j] acquired-all-acquired [of True ?take-sb\_j O\_j]
all-acquired-append [of ?take-sb\_j ?drop-sb\_j]
have acquired True ?take-sb\_j O\_j \cup all-acquired ?drop-sb\_j \subseteq O\_j \cup all-acquired sb\_j
have a ∈ read-only-reads (acquired True ?take-sbj Oj) ?drop-sbj
using a-owned ownership-distinct [OF i-bound j-bound neq-i-j ts\_sb-\_i ts\_sb-\_j]
by auto

with read-only-reads-unowned [OF j-bound i-bound neq-i-j [symmetric] ts\_sb-\_j ts\_sb-\_i]
a-owned

show False
by auto

qed

} note a-notin-unforwarded-non-volatile-reads-drop = this

have valid-reads': valid-reads m\_sb' ts\_sb'

proof (unfold-locales)
fix j p j isj Oj Rj Dj \_j sbj
assume j-bound: j < length ts\_sb'
assume ts-j: ts\_sb' !j = (p\_j, is\_j, j, sbj, D\_j, O_j, R_j)

show reads-consistent False O_j m\_sb' sbj

proof (cases i=j)

next

from reads-consis ts-j j-bound sb show ?thesis

by (clarsimp simp add: True m\_sb' Write\_sb ts\_sb' O\_sb' False reads-consistent-pending-write-antimono)

next

from j-bound have j-bound': j < length ts\_sb
by (simp add: ts\_sb')

moreover from ts-j False have ts-j': ts\_sb' !j = (p\_j, is\_j, j, sbj, D\_j, O_j, R_j)
using j-bound by (simp add: ts\_sb')

ultimately have consis-m: reads-consistent False O_j m\_sb sbj
by (rule valid-reads)

from a-unowned-by-others [rule-format, OF j-bound' False] ts-j'

have a-unowned:a \notin O\_j \cup all-acquired sbj
by simp

let ?take-sbj = takeWhile (Not \circ is-volatile-Write\_sb) sbj
let ?drop-sbj = dropWhile (Not \circ is-volatile-Write\_sb) sbj

from a-unowned acquired-reads-all-acquired [of True ?take-sbj O_j]
all-acquired-append [of ?take-sbj ?drop-sbj]

have a-not-acq-reads: a \notin acquired-reads True ?take-sbj O_j
by auto

moreover

note a-unfw= a-notin-unforwarded-non-volatile-reads-drop [OF j-bound' ts-j' False]

ultimately

show ?thesis
\begin{verbatim}
using \textnormal{reads-consistent-mem}-eq-on-unforwarded-non-volatile-reads-drop \quad \textbf{where} W=\{} \text{and} A=\text{unforwarded-non-volatile-reads} \ \text{?drop-sb} \ \text{\{} \text{\cup} \text{acquired-reads} \ \text{True} \ \text{?take-sb} \ \text{O}_j \ \text{and} \\
\text{m}'=(m_{sb}(a:=v)), \ \text{OF} \ - \ - \ \text{consis-m} \\
\text{by (fastforce simp add: m_{sb}')} \\
\text{qed} \\
\text{qed}

\textbf{have} valid-own'': valid-ownership $S_{sb}' \ ts_{sb}'$
\textbf{proof} (intro-locales)
\textbf{show} outstanding-non-volatile-refs-owned-or-read-only $S_{sb}' \ ts_{sb}'$
\textbf{proof} –
\textbf{from} outstanding-non-volatile-refs-owned-or-read-only \ [OF i-bound ts_{sb}-i] \ sb
\textbf{have} non-volatile-owned-or-read-only \text{False} $S_{sb} \ O_{sb} \ sb$
\textbf{by (auto simp add: Write_{sb} False)}
\textbf{from} outstanding-non-volatile-refs-owned-or-read-only-nth-update \ [OF i-bound this]
\textbf{show} \ ?thesis \textbf{by (simp add: ts_{sb}' Write_{sb} False O_{sb}' S_{sb}' )} \\
\text{qed}
\textbf{next}
\textbf{show} outstanding-volatile-writes-unowned-by-others $ts_{sb}'$
\textbf{proof} –
\textbf{from} sb
\textbf{have} out: outstanding-refs is-volatile-Write_{sb} $sb' \subseteq$ outstanding-refs is-volatile-Write_{sb} \\
\text{sb}
\textbf{by (auto simp add: Write_{sb} False)}
\textbf{have} acq: all-acquired $sb' \subseteq$ all-acquired $sb$
\textbf{by (auto simp add: Write_{sb} False sb)}
\textbf{from} outstanding-volatile-writes-unowned-by-others-store-buffer \ [OF i-bound ts_{sb}-i out acq]
\textbf{show} \ ?thesis \textbf{by (simp add: ts_{sb}' Write_{sb} False O_{sb}' S_{sb}' )} \\
\text{qed}
\textbf{next}
\textbf{show} read-only-reads-unowned $ts_{sb}'$
\textbf{proof} –
\textbf{have} ro: read-only-reads \ (acquired True (takeWhile (Not \circ is-volatile-Write_{sb}) \ sb')) $O_{sb}$
\textbf{by (auto simp add: Write_{sb} False)}
\textbf{have} acq: all-acquired $sb' \subseteq$ all-acquired $sb$
\textbf{by (auto simp add: Write_{sb} False sb)}
\textbf{from} read-only-reads-unowned-nth-update \ [OF i-bound ts_{sb}-i ro this]
\textbf{show} \ ?thesis \textbf{by (simp add: ts_{sb}' Write_{sb} False O_{sb}' S_{sb}' )} \\
\text{qed}
\textbf{next}
\textbf{show} ownership-distinct $ts_{sb}'$
\textbf{proof} –
\textbf{have} acq: all-acquired $sb' \subseteq$ all-acquired $sb$
\end{verbatim}
by (auto simp add: Write sb False sb)
with ownership-distinct-instructions-read-value-store-buffer-independent
[OF i-bound ts sb-i]
show ?thesis by (simp add: ts sb Write sb False O sb)
qed
qed

have valid-sharing'; valid-sharing S sb' ts sb'
proof (intro-locales)
  from outstanding-non-volatile-writes-unshared [OF i-bound ts sb-i]
  have non-volatile-writes-unshared S sb sb'
    by (auto simp add: sb Write sb False)
  from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
  show outstanding-non-volatile-writes-unshared S sb' ts sb'
    by (simp add: ts sb' S sb'
  next
  from sharing-consis [OF i-bound ts sb-i]
  have sharing-consistent S sb O sb sb'
    by (auto simp add: sb Write sb False)
  from sharing-consis-nth-update [OF i-bound this]
  show sharing-consis S sb' ts sb'
    by (simp add: ts sb' O sb' S sb')
  next
  from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts sb-i]
    show read-only-unowned S sb' ts sb'
      by (simp add: S sb' ts sb' O sb'
  next
  from unowned-shared-nth-update [OF i-bound ts sb-i subset-refl]
  show unowned-shared S sb' ts sb'
    by (simp add: ts sb' O sb' S sb')
  next
  from no-outstanding-write-to-read-only-memory [OF i-bound ts sb-i]
  have no-write-to-read-only-memory S sb sb'
    by (auto simp add: sb Write sb False)
  from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
  show no-outstanding-write-to-read-only-memory S sb' ts sb'
    by (simp add: S sb' ts sb' sb)
qed

from is-sim
obtain is-sim: instrs (dropWhile (Not o is-volatile-Write sb) sb') @ is sb =
  is @ prog-instrs (dropWhile (Not o is-volatile-Write sb) sb')
by (simp add: suspends sb Write sb False)

have (ts,m,S) ⇒_d* (ts,m,S) by blast

moreover
note flush-commute =
flush-all-until-volatile-write-Write_{sb} non-volatile-commute [OF i-bound ts_{sb}-i [simplified sb Write_{sb} non-vol]]
outstanding-refs-is-Write_{sb} takeWhile disj a-notin-others

note share-commute =
share-all-until-volatile-write-update-sb of sb False, simplified sb
have (ts_{sb} \[[i := (p_{sb}, i_{sb}, \emptyset_{sb}, sb', D_{sb}, O_{sb}, R_{sb})], m_{sb}(a := v), S_{sb}'\]
\sim
(ts_m, m_S)
apply (rule sim-config intros)
apply (simp add: m flush-commute)
apply (clarsimp simp add: S S_{sb}' share-commute)
using leq
apply simp
using i-bound i-bound' is-sim ts-i ts-sim D
apply (clarsimp simp add: Let-def nth-list-update suspends sb False S sb' split: if-split-asm )
done

ultimately
show ?thesis
using valid-own' valid-hist ' valid-reads' valid-sharing' tmps-distinct' m_{sb}'
valid-dd' valid-sops' load-tmps-fresh' enough-flushs' valid-program-history' valid'
t_{sb}', O_{sb}', S_{sb}', R_{sb}'
by (auto simp del: fun-upd-apply)
qed
next
case (Read_{sb} volatile a t v)
from flush this obtain m_{sb}: m_{sb}' = m_{sb} and
O_{sb}': O_{sb}' = O_{sb} and S_{sb}': S_{sb}' = S_{sb} and
R_{sb}': R_{sb}' = R_{sb}
by cases (auto simp add: sb)

have valid-own': valid-ownership S_{sb}', t_{sb}'
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only S_{sb}' t_{sb}'
proof
from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound t_{sb}-i] sb
have non-volatile-owned-or-read-only False S_{sb} O_{sb} sb'
by (auto simp add: Read_{sb})
from outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this]
show ?thesis by (simp add: t_{sb}' Read_{sb} O_{sb}' S_{sb}')
qed
next
show outstanding-volatile-writes-unowned-by-others t_{sb}'
proof
from sb
have out: outstanding-refs is-volatile-Write_{sb} sb' \subseteq outstanding-refs is-volatile-Write_{sb} sb
by (auto simp add: \(\text{Read}_{sb}\))

have acq: all-acquired \(sb'\) \(\subseteq\) all-acquired \(sb\)
  by (auto simp add: \(\text{Read}_{sb}\) \(sb\))

from outstanding-volatile-writes-unowned-by-others-store-buffer 
[OF i-bound ts\(_{sb}\)-i out acq]

show ?thesis by (simp add: ts\(_{sb}\) \(\text{Read}_{sb}\) \(O_{sb}\))

qed

next

show read-only-reads-unowned ts\(_{sb}\)'

proof –

have ro: read-only-reads (acquired True (takeWhile (Not \(\circ\) is-volatile-Write\(_{sb}\)) \(sb')\) \(O_{sb}\)) 
  (dropWhile (Not \(\circ\) is-volatile-Write\(_{sb}\)) \(sb')\) 
  \(\subseteq\) read-only-reads (acquired True (takeWhile (Not \(\circ\) is-volatile-Write\(_{sb}\)) \(sb\)) \(O_{sb}\)) 
  (dropWhile (Not \(\circ\) is-volatile-Write\(_{sb}\)) \(sb\))
  by (auto simp add: \(sb\) \(\text{Read}_{sb}\))

have \(O_{sb} \cup\) all-acquired \(sb'\) \(\subseteq\) \(O_{sb}\) \(\cup\) all-acquired \(sb\)
  by (auto simp add: \(sb\) \(\text{Read}_{sb}\))

from read-only-reads-unowned-nth-update [OF i-bound ts\(_{sb}\)-i ro this]

show ?thesis by (simp add: ts\(_{sb}\) \(sb\) \(O_{sb}\))

qed

next

show ownership-distinct ts\(_{sb}\)'

proof –

have acq: all-acquired \(sb'\) \(\subseteq\) all-acquired \(sb\)
  by (auto simp add: \(\text{Read}_{sb}\) \(sb\))

with ownership-distinct-instructions-read-value-store-buffer-independent 
[OF i-bound ts\(_{sb}\)-i]

show ?thesis by (simp add: ts\(_{sb}\) \(\text{Read}_{sb}\) \(O_{sb}\))

qed

qed

have valid-sharing': valid-sharing \(S_{sb}'\) ts\(_{sb}'\)

proof (intro-locales)

from outstanding-non-volatile-writes-unshared [OF i-bound ts\(_{sb}\)-i]

have non-volatile-writes-unshared \(S_{sb}\) sb'
  by (auto simp add: \(sb\) \(\text{Read}_{sb}\))

from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]

show outstanding-non-volatile-writes-unshared \(S_{sb}'\) ts\(_{sb}'\)
  by (simp add: ts\(_{sb}'\) \(S_{sb}'\))

next

from sharing-consis [OF i-bound ts\(_{sb}\)-i]

have sharing-consistent \(S_{sb}\) \(O_{sb}\) sb'
  by (auto simp add: \(sb\) \(\text{Read}_{sb}\))

from sharing-consis-nth-update [OF i-bound this]

show sharing-consis \(S_{sb}'\) ts\(_{sb}'\)
  by (simp add: ts\(_{sb}'\) \(O_{sb}'\) \(S_{sb}'\))

next

from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts\(_{sb}\)-i]]
show read-only-unowned $S_{sb}' \cap t_{sb}'$
by (simp add: $S_{sb}' \cap t_{sb}' \cap O_{sb}'$)

next
from unowned-shared-nth-update [OF i-bound $t_{sb}-i \subset \text{subset-refl}$]
show unowned-shared $S_{sb}' \cap t_{sb}'$
by (simp add: $t_{sb}' \cap O_{sb}' \cap S_{sb}'$)

next
from no-outstanding-write-to-read-only-memory [OF i-bound $t_{sb}-i$]
have no-write-to-read-only-memory $S_{sb} \cap t_{sb}'$
by (auto simp add: $sb \cap \text{Read}_{sb}$)
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
show no-outstanding-write-to-read-only-memory $S_{sb}' \cap t_{sb}'$
by (simp add: $S_{sb}' \cap t_{sb}' \cap sb$)
qed

have valid-reads': valid-reads $m_{sb}' \cap t_{sb}'$
proof –
from valid-reads [OF i-bound $t_{sb}-i$]
have reads-consistent False $O_{sb} \cap m_{sb} \cap sb'$
by (simp add: sb \text{Read}_{sb})
from valid-reads-nth-update [OF i-bound this]
show \text{thesis} by (simp add: $m_{sb}' \cap t_{sb}' \cap O_{sb}'$)
qed

have valid-program-history': valid-program-history $t_{sb}'$
proof –
from valid-program-history [OF i-bound $t_{sb}-i$]
have causal-program-history $is_{sb} \cap sb$.
then have causal': causal-program-history $is_{sb}' \cap sb'$
by (simp add: sb \text{causal-program-history-def})

from valid-last-prog [OF i-bound $t_{sb}-i$]
have last-prog $p_{sb} \cap sb = p_{sb}$.
\text{hence} last-prog $p_{sb} \cap sb' = p_{sb}$
by (simp add: sb \text{Read}_{sb})

from valid-program-history-nth-update [OF i-bound causal' this]
show \text{thesis}
by (simp add: $t_{sb}'$)
qed

from is-sim
have is-sim: \text{instrs} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb') \cap is_{sb} = \text{is} \cap \text{prog-instrs} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb')
by (simp add: sb \text{Read}_{sb} \text{suspends})

from valid-history [OF i-bound $t_{sb}-i$]
have $\theta_{sb} \cap v \times \theta_{sb} t = \text{Some} \ v$
by (simp add: history-consistent-access-last-read sb \text{Read}_{sb} \text{split:option.splits})
have \((\text{ts}, m, S) \Rightarrow d^* (\text{ts}, m, S)\) by blast

moreover

**note** flush-commute = flush-all-until-volatile-write-Readsb-commute\([\text{OF } i\text{-bound } \text{ts}_{sb}\text{-i }\text{[simplified sb } \text{Read}_{sb}\])

**note** share-commute =
share-all-until-volatile-write-update-sb\([\text{of sb'} sb, \text{OF } i\text{-bound } \text{ts}_{sb}\text{-i, simplified sb Read}_{sb}, \text{simplified}]\)

have \((\text{ts}_{sb} [i := (p_{sb}, \text{is}_{sb}, \theta_{sb}, \text{sb'}, D_{sb}, O_{sb}, R_{sb'}), m_{sb}, S_{sb'}]) \sim (\text{ts}, m, S)\)
apply (rule sim-config,intros)
apply (simp add: m flush-commute)
apply (clarsimp simp add: SS_{sb'} share-commute)
using leq
apply simp
using i-bound i-bound' ts-sim ts-i is-sim \(D\)
apply (clarsimp simp add: Let-def nth-list-update sb suspends Read_{sb} S_{sb'} R_{sb'}
split: if-split-asm)
done

ultimately show ?thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' \(m_{sb'}\)
valid-dd' valid-sops' load-tmps-fresh' enough-flushs' valid-sharing'
valid-program-history' valid'
\(ts_{sb'} O_{sb'} S_{sb'}\)
by (auto simp del: fun-upd-apply)
next
case (Prog_{sb} p_1 p_2 mis)
from flush this obtain \(m_{sb'} : m_{sb'} = m_{sb} \text{ and } O_{sb'} = O_{sb} \text{ and } S_{sb'} = S_{sb} \text{ and } R_{sb'} = R_{sb}\)
by cases (auto simp add: sb)

have valid-own': valid-ownership \(S_{sb'} ts_{sb'}\)
proof (intro-locale)
show outstanding-non-volatile-refs-owned-or-read-only \(S_{sb'} ts_{sb'}\)
proof
from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb}-i] sb
have non-volatile-owned-or-read-only False \(S_{sb} O_{sb} sb'\)
by (auto simp add: Prog_{sb})
from outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb} Prog_{sb} O_{sb'} S_{sb'})
qed
next
show outstanding-volatile-writes-unowned-by-others \(ts_{sb'}\)
proof
from sb

667
have out: outstanding-refs is-volatile-Write_{sb} sb' ⊆ outstanding-refs is-volatile-Write_{sb} sb
by (auto simp add: Prog_{sb})
have acq: all-acquired sb' ⊆ all-acquired sb
by (auto simp add: Prog_{sb} sb)
from outstanding-volatile-writes-unowned-by-others-store-buffer [OF i-bound ts_{sb}-i out acq]
show ?thesis by (simp add: ts_{sb}′ Prog_{sb} O_{sb}′)
qed
next
show read-only-reads-unowned ts_{sb}'
proof -
have ro: read-only-reads (acquired True (takeWhile (Not ◦ is-volatile-Write_{sb}) sb') O_{sb})
  (dropWhile (Not ◦ is-volatile-Write_{sb}) sb')
  ⊆ read-only-reads (acquired True (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) O_{sb})
  (dropWhile (Not ◦ is-volatile-Write_{sb}) sb)
  by (auto simp add: sb Prog_{sb})
have O_{sb} ⊆ all-acquired sb' ⊆ O_{sb} ⊆ all-acquired sb
  by (auto simp add: sb Prog_{sb})
from read-only-reads-unowned-nth-update [OF i-bound ts_{sb}-i ro this]
show ?thesis
  by (simp add: ts_{sb}′ sb O_{sb}′)
qed
next
show ownership-distinct ts_{sb}'
proof -
have acq: all-acquired sb' ⊆ all-acquired sb
  by (auto simp add: Prog_{sb} sb)
with ownership-distinct-instructions-read-value-store-buffer-independent [OF i-bound ts_{sb}-i]
show ?thesis
  by (simp add: ts_{sb}′ sb O_{sb}′)
qed

have valid-sharing': valid-sharing S_{sb}' ts_{sb}'
proof (intro-locales)
from outstanding-non-volatile-writes-unshared [OF i-bound ts_{sb}-i]
have non-volatile-writes-unshared S_{sb} sb'
  by (auto simp add: sb Prog_{sb})
from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
show outstanding-non-volatile-writes-unshared S_{sb}' ts_{sb}'
  by (simp add: ts_{sb} ′ S_{sb}′)
next
from sharing-consis [OF i-bound ts_{sb}-i]
have sharing-consistent S_{sb} O_{sb} sb'
  by (auto simp add: sb Prog_{sb})
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis S_{sb}' ts_{sb}'
  by (simp add: ts_{sb}′ O_{sb}′ S_{sb}′)
next
from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound ts\_sb\_i]]

show read-only-unowned \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( S_{sb}' \) \( ts_{sb}' \) \( O_{sb}' \))

next

from unowned-shared-nth-update [OF i-bound ts\_sb\_i subset-refl]

show unowned-shared \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( ts_{sb}' \) \( O_{sb}' \) \( S_{sb}' \))

next

from no-outstanding-write-to-read-only-memory [OF i-bound ts\_sb\_i]

have no-write-to-read-only-memory \( S_{sb} \)
  by (auto simp add: sb Prog)

from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]

show no-outstanding-write-to-read-only-memory \( S_{sb}' \) \( ts_{sb}' \)
  by (simp add: \( S_{sb}' \) \( ts_{sb}' \) \( sb \))

qed

have valid-reads': valid-reads \( m_{sb}' \) \( ts_{sb}' \)
  proof –

from valid-reads [OF i-bound ts\_sb\_i]

have reads-consistent False \( O_{sb} \) \( m_{sb} \) \( sb \)
  by (simp add: sb Prog)

from valid-reads-nth-update [OF i-bound this]

show ?thesis by (simp add: \( m_{sb}' \) \( ts_{sb}' \) \( O_{sb}' \))

qed

have valid-program-history': valid-program-history \( ts_{sb}' \)
  proof –

from valid-program-history [OF i-bound ts\_sb\_i]

have causal-program-history \( is_{sb} \) \( sb \).

then have causal': causal-program-history \( is_{sb} \) \( sb' \)
  by (simp add: sb Prog\_sb causal-program-history-def)

from valid-last-prog [OF i-bound ts\_sb\_i]

have last-prog \( p_{sb} \) \( sb \) = \( p_{sb} \).

hence last-prog \( p_{2} \) \( sb' \) = \( p_{sb} \)
  by (simp add: sb Prog\_sb)

from last-prog-to-last-prog-same [OF this]

have last-prog \( p_{sb} \) \( sb' \) = \( p_{sb} \).

from valid-program-history-nth-update [OF i-bound causal' this]

show ?thesis
  by (simp add: \( ts_{sb}' \))

qed

from is-sim

have is-sim: \( \text{instrs} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_sb) \) \( sb') \) @ \( is_{sb} \) = is @ \( \text{prog-instrs} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_sb) \) \( sb') \)
  by (simp add: suspends sb Prog\_sb)

669
have \((ts,m,S) \Rightarrow d^* (ts,m,S)\) by blast

moreover

note flush-commute = flush-all-until-volatile-write-Prog\sb-commute [OF i-bound ts\sb-i [simplified \sb \text{Prog} \sb]]

note share-commute = share-all-until-volatile-write-update-\sb \text{[of sb'} \sb, \text{OF - i-bound ts}\sb-i, \text{simplified sb Prog} \sb, \text{simplified}]

have \((ts\sb[i := (p\sb, is\sb, \alpha\sb, D\sb, O\sb, R\sb, S\sb')], m\sb, S\sb') \sim (ts, m, S)\)
apply (rule sim-config.intros)
apply (simp add: m flush-commute)
apply (clarsimp simp add: \sb Prog\sb, simplified)
using leq
apply simp
using i-bound i-bound' ts-sim ts-i is-sim D
apply (clarsimp simp add: Let-def nth-list-update \sb R\sb', \sb Suspends Prog\sb, simplified)
done
ultimately show \(?\text{thesis}\)
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' m\sb' valid-dd' valid-sops' load-tmps-fresh' enough-flushs' valid-sharing'
valid-program-history' valid' ts\sb' S\sb' O\sb' R\sb' S\sb'
by (auto simp del: fun-upd-apply)

next
case (Ghost\sb A L R W)
from flush Ghost\sb
obtain
\O\sb': \O\sb' = \O\sb \cup A - R and
\S\sb': \S\sb' = \S\sb \oplus W R \oplus_A L and
\R\sb': \R\sb' = \text{augment-rels (dom S\sb) R R\sb and}
m\sb': \m\sb' = \m\sb
by cases (auto simp add: \sb)

from sharing-consis [OF i-bound ts\sb-i]
obtain
A-shared-owned: A \subseteq \text{dom S\sb} \cup O\sb and
L-subset: L \subseteq A and
A-R: A \cap R = \{\} and
R-owned: R \subseteq O\sb
by (clarsimp simp add: \sb Ghost\sb)

have valid-own': valid-ownership S\sb' ts\sb'
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only S\sb' ts\sb'
proof

670
fix j is\_j O\_j R\_j D\_j acq\_j \hat{\alpha}_j sb\_j p\_j
assume j-bound: j < length ts\_sb\' 
assume ts\_sb\' \_j: ts\_sb \_j = (p\_j, is\_j, \hat{\alpha}_j, sb\_j, D\_j, O\_j, R\_j)
show non-volatile-owned-or-read-only False S\_sb\' O\_j sb\_j
proof (cases j=i)
  case True
  from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts\_sb\_-i]
  have non-volatile-owned-or-read-only False (S\_sb \oplus W R \ominus A L) (O\_sb \cup A \ominus R) sb\' 
  by (auto simp add: sb Ghost\_sb non-volatile-owned-or-read-only-pending-write-antimono)
  then show ?thesis
    using True i-bound ts\_sb\_' \_j 
    by (auto simp add: ts\_sb\_' S\_sb\_' sb\_O\_sb\_')
next
  case False
  from j-bound
  have j-bound\_': j < length ts\_sb
  by (auto simp add: ts\_sb\_' j-bound)
  with ts\_sb\_j False i-bound
  have ts\_sb\_j: ts\_sb \_j = (p\_j, is\_j, \hat{\alpha}_j, sb\_j, D\_j, O\_j, R\_j)
  by (auto simp add: ts\_sb\_' j-bound)
  note nvo = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound\_\' ts\_sb\_-j]
  from read-only-unowned [OF i-bound ts\_sb\_-i] R-owned
  have R \cap read-only S\_sb = {}
  by auto
  with read-only-reads-unowned [OF j-bound\_\' i-bound False ts\_sb\_-j ts\_sb\_-i] L-subset
  have \forall a \in read-only S\_sb \rightarrow a \in read-only (S\_sb \oplus W R \ominus A L )
  by (auto simp add: in-read-only-convs sb Ghost\_sb)
  from non-volatile-owned-or-read-only-reads-eq\_\' [OF nvo this]
  have non-volatile-owned-or-read-only False (S\_sb \oplus W R \ominus A L) O\_j sb\_j,
  thus ?thesis by (simp add: S\_sb\_')
qed
cqed

next
show outstanding-volatile-writes-unowned-by-others ts\_sb\'
proof (unfold-locales)
  fix i\_1 p\_1 s\_1 O\_1 R\_1 D\_1 xs\_1 sb\_1 p\_j is\_j O\_j R\_j D\_j xs\_j sb\_j
  assume i\_1-bound: i\_1 < length ts\_sb\'
  assume j-bound: j < length ts\_sb
  assume i\_1\_j: i\_1 \neq j
  assume ts\_i\_1: ts\_sb\'_i\_1 = (p\_1, is\_1, xs\_1, sb\_1, D\_1, O\_1, R\_1)
  assume ts\_j: ts\_sb\'_j = (p\_j, is\_j, xs\_j, sb\_j, D\_j, O\_j, R\_j)
  show (O\_j \cup all-acquired sb\_j) \cap outstanding-refs is-volatile-Write\_sb sb\_1 = {}
proof (cases i\_1\_i)
  case True
from i1-j True have neq-i-j: i ≠ j
by auto
from j-bound have j-bound': j < length ts_{sb}
by (simp add: ts_{sb}')
from ts-j neq-i-j have ts-j': ts_{sb}'{j} = (p_{j}, is_{j}, xs_{j}, sb_{j}, D_{j}, O_{j}, R_{j})
by (simp add: ts_{sb}')
from outstanding-volatile-writes-unowned-by-others [OF i-bound j-bound' neq-i-j ts_{sb}'-i ts-j'] ts-i1 i-bound ts_{sb}'-i True show ?thesis
by (clarsimp simp add: ts_{sb}' sb Ghost_{sb})
next
case False
note i1-i = this
from i1-bound have i1-bound': i1 < length ts_{sb}
by (simp add: ts_{sb}')
hence i1-bound'': i1 < length (map owned ts_{sb})
by auto
from ts-i1 False have ts-i1': ts_{sb}'!i1 = (p_{i1}, is_{i1}, xs_{i1}, sb_{i1}, D_{i1}, O_{i1}, R_{i1})
by (simp add: ts_{sb}' sb)
show ?thesis
proof (cases j=i)
case True
from outstanding-volatile-writes-unowned-by-others [OF i1-bound' i-bound i1-i ts-i1' ts_{sb}'-i]
have (O_{sb} ∪ all-acquired sb) ∩ outstanding-refs is-volatile-Write_{sb} sb_{1} = {}.
then show ?thesis
using True i1-i ts-j ts_{sb}'-i i-bound
by (auto simp add: sb Ghost_{sb} ts_{sb}' O_{sb}')
next
case False
from j-bound have j-bound': j < length ts_{sb}
by (simp add: ts_{sb}')
from ts-j False have ts-j': ts_{sb}'{j} = (p_{j}, is_{j}, xs_{j}, sb_{j}, D_{j}, O_{j}, R_{j})
by (simp add: ts_{sb}')
from outstanding-volatile-writes-unowned-by-others [OF i1-bound' j-bound' i1-j ts-i1' ts-j']
show (O_{j} ∪ all-acquired sb_{j}) ∩ outstanding-refs is-volatile-Write_{sb} sb_{1} = {}.
qed
qed
qed
next
show read-only-reads-unowned ts_{sb}'
proof
fix n m
fix p_{n} is_{n} O_{n} R_{n} D_{n} ∅_{n} sb_{n} p_{m} is_{m} O_{m} R_{m} D_{m} ∅_{m} sb_{m}
assume n-bound: n < length ts_{sb}'
and m-bound: m < length ts_{sb}'
and neq-n-m: n ≠ m
and nth: ts_{sb}'{n} = (p_{n}, is_{n}, ∅_{n}, sb_{n}, D_{n}, O_{n}, R_{n})
and nth: ts_{sb}'{m} = (p_{m}, is_{m}, ∅_{m}, sb_{m}, D_{m}, O_{m}, R_{m})
from n-bound have n-bound': n < length ts_{sb} by (simp add: ts_{sb}')
from m-bound have m-bound': m < length ts\sb' by (simp add: ts\sb')

show (O\m \union all-acquired sb\m) \inter
  read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write\sb) sb\n) O\n)
  (dropWhile (Not \circ is-volatile-Write\sb) sb\n) = 
  
proof (cases m=i)
  case True
  with neq-n-m have neq-n-i: n # i
  by auto
  with n-bound nth i-bound have nth': ts\sb!n = (p\n, is\n, \emptyset\n, sb\n, D\n, O\n, R\n)
  by (auto simp add: ts\sb'
)
  note read-only-reads-unowned [OF n-bound' i-bound neq-n-i nth' ts\sb-i]
  then
  show \?thesis
    using True ts\sb-i neq-n-i nth m-bound' m-bound' L-subset
    by (auto simp add: ts\sb' O\sb' Ghost\sb)
next
  case False
  note neq-m-i = this
  with m-bound mth i-bound have mth': ts\sb!m = (p\m, is\m, \emptyset\m, sb\m, D\m, O\m, R\m)
  by (auto simp add: ts\sb'
)
  show \?thesis
proof (cases n=i)
  case True
  from read-only-reads-append [of (O\sb \union A - R) (takeWhile (Not \circ is-volatile-Write\sb) sb\n) (dropWhile (Not \circ is-volatile-Write\sb) sb\n)]
  have read-only-reads
    (acquired True (takeWhile (Not \circ is-volatile-Write\sb) sb\n) (O\sb \union A - R))
    (dropWhile (Not \circ is-volatile-Write\sb) sb\n) \subseteq read-only-reads (O\sb \union A - R)
  sbn
  by auto
  with ts\sb-i nth mth neq-m-i m-bound' True
  read-only-reads-unowned [OF i-bound m-bound' False [symmetric] ts\sb-i mth']
  show \?thesis
  by (auto simp add: ts\sb' sb O\sb' Ghost\sb)
next
  case False
  with n-bound nth i-bound have nth': ts\sb!n = (p\n, is\n, \emptyset\n, sb\n, D\n, O\n, R\n)
  by (auto simp add: ts\sb')
  from read-only-reads-unowned [OF n-bound' m-bound' neq-n-m nth' mth'] False
  neq-m-i
  show \?thesis
  by (clarsimp)
  qed
  qed
  qed
next

673
show ownership-distinct $ts_{sb}'$

proof (unfold-locales)

fix $i_1 \; p_1 \; s_1 \; O_1 \; R_1 \; D_1 \; x_1 \; sb_1 \; p_j \; s_j \; O_j \; R_j \; D_j \; x_j \; sb_j$

assume $i_1$-bound: $i_1 < \text{length } ts_{sb}'$
assume $j$-bound: $j < \text{length } ts_{sb}'$
assume $i_1$-$j$: $i_1 \neq j$
assume $ts_i$: $ts_{sb}!i_1 = (p_1, i_1, x_1, sb_1, D_1, O_1, R_1)$
assume $ts_j$: $ts_{sb}!j = (p_j, i_j, x_j, sb_j, D_j, O_j, R_j)$
show $(O_1 \cup \text{all-acquired } sb_1) \cap (O_j \cup \text{all-acquired } sb_j) = \{\}$

proof (cases $i_1 = i$)
  case True
  with $i_1$-$j$ have $i$-$j$: $i \neq j$
  by simp

  from $j$-bound have $j$-bound': $j < \text{length } ts_{sb}$
  by (simp add: $ts_{sb}'$)
  hence $j$-bound'': $j < \text{length (map owned } ts_{sb})$
  by simp
  from $ts_j$ $i$-$j$ have $ts_j$: $ts_{sb}!j = (p_j, i_j, x_j, sb_j, D_j, O_j, R_j)$
  by (simp add: $ts_{sb}'$)

  from ownership-distinct [OF $i$-bound $j$-bound' $i$-$j$ $ts_{sb}$-$i$]
  show $?thesis$
  using $ts_{sb}$-$i$ True $ts_i$ $i$-bound $O_{sb}'$
  by (auto simp add: $ts_{sb}'$ $sb$ Ghost$_{sb}$)

next
  case False
  note $i_1$-$i = \text{this}$
  from $i_1$-bound have $i_1$-bound': $i_1 < \text{length } ts_{sb}$
  by (simp add: $ts_{sb}'$)
  hence $i_1$-bound'': $i_1 < \text{length (map owned } ts_{sb})$
  by simp
  from $ts_i$ False have $ts_i$: $ts_{sb}!i_1 = (p_1, i_1, x_1, sb_1, D_1, O_1, R_1)$
  by (simp add: $ts_{sb}'$)
  show $?thesis$
  proof (cases $j$=$i$)
    case True
    from ownership-distinct [OF $i_1$-bound' $i$-bound' $i_1$-$i$ $ts_{sb}$-$i$]
    show $?thesis$
    using $ts_{sb}$-$i$ True $ts_j$ $i$-bound $O_{sb}'$
    by (auto simp add: $ts_{sb}'$ $sb$ Ghost$_{sb}$)

next
  case False
  from $j$-bound have $j$-bound': $j < \text{length } ts_{sb}$
  by (simp add: $ts_{sb}'$)
  from $ts_j$ False have $ts_j$: $ts_{sb}!j = (p_j, i_j, x_j, sb_j, D_j, O_j, R_j)$
  by (simp add: $ts_{sb}'$)
  from ownership-distinct [OF $i_1$-bound' $j$-bound' $i_1$-$j$ $ts_{sb}$-$i$]
  show $?thesis$
  qed
have valid-sharing\textquotesingle{}: valid-sharing ($S_{sb} \oplus_W R \ominus_A L$) $ts_{sb}'$

proof (intro-locales)
show outstanding-non-volatile-writes-unshared ($S_{sb} \oplus_W R \ominus_A L$) $ts_{sb}'$

proof (unfold-locales)
fix $j$ $p_j$ $is_j$ $R_j$ $acq_j$ $xs_j$ $sb_j$
assume $j$-bound: $j < \text{length } ts_{sb}'$
assume $j$th: $ts_{sb}'$ $! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)$
show non-volatile-writes-unshared ($S_{sb} \oplus_W R \ominus_A L$) $sb_j$

proof (cases $i=j$)
case True
with outstanding-non-volatile-writes-unshared [OF $i$-bound $ts_{sb}$-i]
i-bound $j$th $ts_{sb}$-i show ?thesis
by (clarsimp simp add: $ts_{sb}'$ $sb$ Ghost$_{sb}$)

next
case False
from $j$-bound have $j$-bound\textquotesingle{}: $j < \text{length } ts_{sb}$
by (auto simp add: $ts_{sb}'$)
from $j$th False have $j$th\textquotesingle{}: $ts_{sb}'$ $! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)$
by (auto simp add: $ts_{sb}'$)
from $j$-bound $j$th $i$-bound False
have $j$: non-volatile-writes-unshared $S_{sb}$ $sb_j$
apply
apply (rule outstanding-non-volatile-writes-unshared)
apply (auto simp add: $ts_{sb}'$)
done
from $j$th False have $j$th\textquotesingle{}: $ts_{sb}'$ $! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)$
by (auto simp add: $ts_{sb}'$)
from outstanding-non-volatile-writes-unshared [OF $j$-bound\textquotesingle{} $j$th\textquotesingle{}]
have unshared: non-volatile-writes-unshared $S_{sb}$ $sb_j$.

have $\forall a \in \text{dom } (S_{sb} \oplus_W R \ominus_A L) - \text{dom } S_{sb}, a \not\in \text{outstanding-refs is-non-volatile-Write}_{sb}$ $sb_j$

proof –
{
fix $a$
assume a-in: $a \in \text{dom } (S_{sb} \oplus_W R \ominus_A L) - \text{dom } S_{sb}$
hence a-R: $a \in R$
by clarsimp
assume a-in-j: $a \in \text{outstanding-refs is-non-volatile-Write}_{sb}$ $sb_j$
have False
proof –
from non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF
outstanding-non-volatile-refs-owned-or-read-only [OF $j$-bound\textquotesingle{} $j$th\textquotesingle{}]]
a-in-j
have $a \in O_j \cup \text{all-acquired } sb_j$
by auto

675
moreover
with ownership-distinct \([OF\ i\text{-}bound\ j\text{-}bound\ 'False\ ts_{sb}\text{-}i\ jth\ ']\ a\text{-}R\ R\text{-}owned
show\ False
by\ blast
qed

}\)
thus\ ?thesis\ by\ blast
qed

from\ non\text{-}volatile\text{-}writes\text{-}unshared\text{-}no\text{-}outstanding\text{-}non\text{-}volatile\text{-}Write_{sb}
[OF\ unshared\ this]
show\ ?thesis.
qed
qed
next
show\ sharing\text{-}consis\ \((S_{sb}\oplus\mathcal{W}\ R\ominus_{A} L)\)\ ts_{sb}'
proof\ (unfold\-locales)
fix\ j\ p_{j}\ is_{j}\ O_{j}\ R_{j}\ D_{j}\ acq_{j}\ xs_{j}\ sb_{j}
assume\ j\text{-}bound:\ j <\ length\ ts_{sb}'
assume\ jth: ts_{sb}'!\ j = (p_{j},is_{j},xs_{j},sb_{j},D_{j},O_{j},R_{j})
show\ sharing\text{-}consistent\ \((S_{sb}\oplus\mathcal{W}\ R\ominus_{A} L)\)\ O_{j}\ sb_{j}
proof\ (cases\ i=j)
case\ True
with\ i\text{-}bound\ jth\ ts_{sb}\text{-}i\ sharing\text{-}consis\ [OF\ i\text{-}bound\ ts_{sb}\text{-}i]
show\ ?thesis
by\ (clarsimp\ simp\ add: ts_{sb}'\ sb\ Ghost_{sb}\ O_{sb}')
next
case\ False
from\ j\text{-}bound\ have\ j\text{-}bound':\ j <\ length\ ts_{sb}
by\ (auto\ simp\ add: ts_{sb}')
from\ jth\ False\ have\ jth': ts_{sb}'!\ j = (p_{j},is_{j},xs_{j},sb_{j},D_{j},O_{j},R_{j})
by\ (auto\ simp\ add: ts_{sb}')
from\ sharing\text{-}consis\ [OF\ j\text{-}bound'jth']
have\ consis:\ sharing\text{-}consistent\ \(S_{sb}\)\ O_{j}\ sb_{j}.

have\ acq\text{-}cond: all\text{-}acquired\ sb_{j}\cap\ dom\ S_{sb} -\ dom\ \(S_{sb}\oplus\mathcal{W}\ R\ominus_{A} L\) = \{}
proof
{\}
fix\ a
assume\ a\text{-}acq: a \in\ all\text{-}acquired\ sb_{j}
assume\ a \in\ dom\ S_{sb}
assume\ a\text{-}L: a \in L
have\ False
proof
from\ ownership\text{-}distinct\ [OF\ i\text{-}bound\ j\text{-}bound'False\ ts_{sb}\text{-}i\ jth']
have\ A\cap\ all\text{-}acquired\ sb_{j} = \{}
by\ (auto\ simp\ add: sb\ Ghost_{sb})
with a-acq a-L L-subset
show False
by blast
qed

thus ?thesis
by auto
qed

have uns-cond: all-unshared sb_j ∩ dom (S_{sb} ⊕ W R ⊓ A L) − dom S_{sb} = {}
proof –
{
fix a
assume a-uns: a ∈ all-unshared sb_j
assume a ∉ L
assume a-R: a ∈ R
have False
proof –
from unshared-acquired-or-owned [OF consis] a-uns
have a ∈ all-acquired sb_j ∪ O_j by auto
with ownership-distinct [OF i-bound j-bound' False ts_{sb-i} jth'] R-owned a-R
show False
by blast
qed
}

thus ?thesis
by auto
qed

from sharing-consistent-preservation [OF consis acq-cond uns-cond]
show ?thesis
by (simp add: ts_{sb}')
qed

next

show unowned-shared (S_{sb} ⊕ W R ⊓ A L) ts_{sb}'
proof (unfold-locales)
show − ∪ ((λ(_, _, _, O, _). O) ′ set ts_{sb}') ⊆ dom (S_{sb} ⊕ W R ⊓ A L)
proof –

have s: ∪((λ(_, _, _, O, _). O) ′ set ts_{sb}') =
∪((λ(_, _, _, O, _). O) ′ set ts_{sb}) ∪ A − R

apply (unfold ts_{sb}' O_{sb}')
apply (rule acquire-release-ownership-nth-update [OF R-owned i-bound ts_{sb-i}])
apply (rule local.ownership-distinct-axioms)
done

note unowned-shared L-subset A-R
then
show ?thesis
  apply (simp only: s)
  apply auto
  done
qed
qed
next
show read-only-unowned \( (S_{sb} \oplus_W R \ominus_A L) \) \( ts_{sb}' \)
proof
  fix \( j, p \) \( j \) is \( j \) \( O \) \( j \) \( D \) \( j \) \( xs \) \( j \) \( sb \) \( j \)
  assume j-bound: \( j < \) length \( ts_{sb}' \)
  assume jth: \( ts_{sb}' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)
  show \( O_j \cap \) read-only \( (S_{sb} \oplus_W R \ominus_A L) = \{ \} \)
proof (cases i=j)
  case True
  from jth ts_{sb}-i j-bound have \( (O_{sb} \cup A - R) \cap \) read-only \( (S_{sb} \oplus_W R \ominus_A L) = \{ \} \)
  by (auto simp add: in-read-only-convs)
  with jth ts_{sb}-i i-bound True
  show ?thesis
  by (auto simp add: ts_{sb} Ghost_{sb})
next
  case False
  from j-bound have j-bound': \( j < \) length \( ts_{sb} \)
  by (auto simp add: ts_{sb}')
  with jth have jth': \( ts_{sb}' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)
  by (auto simp add: ts_{sb}')
  from read-only-unowned [OF j-bound' jth']
  have \( O_j \cap \) read-only \( S_{sb} = \{ \} \).
  moreover
  from ownership-distinct [OF i-bound j-bound' False ts_{sb}-i jth'] R-owned
  have \( (O_{sb} \cup A) \cap O_j = \{ \} \)
  by (auto simp add: sb Ghost_{sb})
  moreover note R-owned A-R
  ultimately show ?thesis
  by (fastforce simp add: in-read-only-convs split: if-split-asm)
qed
qed
next
show no-outstanding-write-to-read-only-memory \( (S_{sb} \oplus_W R \ominus_A L) \) \( ts_{sb}' \)
proof
  fix \( j, p \) \( j \) is \( j \) \( O \) \( j \) \( R \) \( j \) \( D \) \( j \) \( xs \) \( j \) \( sb \) \( j \)
  assume j-bound: \( j < \) length \( ts_{sb}' \)
  assume jth: \( ts_{sb}' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)
  show no-write-to-read-only-memory \( (S_{sb} \oplus_W R \ominus_A L) \) \( sb_j \)
proof (cases i=j)
  case True
  with jth ts_{sb}-i i-bound show no-write-to-read-only-memory [OF i-bound ts_{sb}-i]
  show ?thesis
  by (auto simp add: sb ts_{sb}' Ghost_{sb})
next
case False
from j-bound have j-bound': j < length ts_{sb}
  by (auto simp add: ts_{sb}')
with False jth have jth': ts_{sb} ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
  by (auto simp add: ts_{sb}')
from no-outstanding-write-to-read-only-memory [OF j-bound']
have nw: no-write-to-read-only-memory S_{sb} sb_j.

have R ∩ outstanding-refs is-Write_{sb} sb_j = {}
proof –
  note dist = ownership-distinct [OF i-bound j-bound' False ts_{sb}-i jth']
  from non-volatile-owned-or-read-only-outstanding-non-volatile-writes
  [OF outstanding-non-volatile-refs-owned-or-read-only [OF j-bound' jth']]
dist
  have outstanding-refs is-non-volatile-Write_{sb} sb_j ∩ O_{sb} = {}
    by auto
  moreover
  from outstanding-volatile-writes-unowned-by-others [OF j-bound' i-bound]
  False [symmetric] jth' ts_{sb}-i ]
  have outstanding-refs is-volatile-Write_{sb} sb_j ∩ O_{sb} = {}
    by (auto simp add: misc-outstanding-refs-convs)
  with R-owned
  show ?thesis by blast
qed

have ∀ a ∈ outstanding-refs is-Write_{sb} sb_j.
  a ∈ read-only (S_{sb} ⊕_{ ∀} R ⊕_{A} L) −→ a ∈ read-only S_{sb}
  by (auto simp add: in-read-only-convs)
from no-write-to-read-only-memory-read-only-reads-eq [OF nw this]
show ?thesis .
qed

have valid-reads': valid-reads m_{sb}' ts_{sb}'
  proof –
  from valid-reads [OF i-bound ts_{sb}-i]
  have reads-consistent False (O_{sb} ∪ A − R) m_{sb} sb'
    by (simp add: sb Ghost_{sb})
  from valid-reads-nth-update [OF i-bound this]
  show ?thesis by (simp add: m_{sb}' ts_{sb}' O_{sb}')
  qed

have valid-program-history': valid-program-history ts_{sb}'
  proof –
  from valid-program-history [OF i-bound ts_{sb}-i]

679
have causal-program-history is_{sb} sb.
then have causal': causal-program-history is_{sb} sb'
  by (simp add: sb Ghost_{sb} causal-program-history-def)

from valid-last-prog [OF i-bound ts_{sb-i}]
have last-prog p_{sb} sb = p_{sb}.
hence last-prog p_{sb} sb' = p_{sb}
  by (simp add: sb Ghost sb)

from valid-program-history-nth-update [OF i-bound causal' this]
show ?thesis
  by (simp add: ts_{sb}'
qed

from is-sim have is-sim: instrs (dropWhile (Not ◦ is-volatile-Write_{sb}') sb) @ is_{sb} =
  is @ prog-instrs (dropWhile (Not ◦ is-volatile-Write_{sb}') sb')
  by (simp add: sb Ghost sb

have (ts,m,S) ⇒ₙ (ts,m,S) by blast
moreover

note flush-commute =
flush-all-until-volatile-write-Ghost_{sb}-commute [OF i-bound ts_{sb-i} [simplified sb Ghost_{sb}]]

have dist-R-L-A: ∀ j p is O R D ∅ sb.
  j < length ts_{sb} → i ≠ j →
  ts_{sb} ! j = (p, is, ∅, sb, D, O, R) → (all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}') sb) ∪
  all-unsafe (takeWhile (Not ◦ is-volatile-Write_{sb}') sb) ∪
  all-acquired (takeWhile (Not ◦ is-volatile-Write_{sb}') sb)) ∩ (R ∪ L ∪ A) = {}
proof -
  { fix j p j is_{j} O_{j} R_{j} D_{j} ∅_{j} sb_{j} x
    assume j-bound: j < length ts_{sb}
    assume neq-i-j: i ≠ j
    assume jth: ts_{sb}[j] = (p_{j}, is_{j}, ∅_{j}, sb_{j}, D_{j}, O_{j}, R_{j})
    assume x-shared: x ∈ all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}') sb_{j}) ∪
      all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}') sb_{j}) ∪
      all-acquired (takeWhile (Not ◦ is-volatile-Write_{sb}') sb_{j})
    assume x-R-L-A: x ∈ R ∪ L ∪ A
    have False
    proof -
      from x-shared all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]
      unshared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]
      all-shared-append [of (takeWhile (Not ◦ is-volatile-Write_{sb}') sb_{j}) (dropWhile
        (Not ◦ is-volatile-Write_{sb}') sb_{j})]
      all-unshared-append [of (takeWhile (Not ◦ is-volatile-Write_{sb}') sb_{j}) (dropWhile
        (Not ◦ is-volatile-Write_{sb}') sb_{j})]

680
all-acquired-append \[\text{of (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j) (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j)}\]

\begin{align*}
\text{have} & \quad x \in \text{all-acquired sb}_j \cup \mathcal{O}_j \\
\text{by} & \quad \text{auto} \\
\text{moreover} & \quad \text{from} \quad x-R-L-A \text{ R-owned L-subset} \\
\text{have} & \quad x \in \text{all-acquired sb}_j \cup \mathcal{O}_{sb} \\
\text{by} & \quad (\text{auto simp add: sb Ghost}_{sb}) \\
\text{moreover} & \quad \text{note ownership-distinct \[\text{OF i-bound j-bound neq-i-j ts}_{sb}-i \ jth\]} \\
\text{ultimately show} & \quad \text{False by blast} \\
\text{qed} & \quad \text{by} \quad \text{blast} \\
\text{thus} & \quad \text{thesis} \quad \text{by} \quad \text{blast} \\
\text{qed} & \quad \text{by}\end{align*}

\begin{align*}
\{ & \quad \text{fix} \quad j \ p_j \ \text{i}s_j \ \mathcal{O}_j \ \mathcal{R}_j \ \mathcal{D}_j \ \emptyset_j \ \text{sb}_j \ x \\
\text{assume} & \quad \text{jth: ts}_{sb}-j = (p_j, i \text{s}_j, \emptyset_j, \text{sb}_j, \mathcal{O}_j, \mathcal{R}_j) \\
\text{assume} & \quad \text{j-bound: } j < \text{length ts}_{sb} \\
\text{assume} & \quad \text{neq: } i \neq j \\
\text{have} & \quad \text{release (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j)} \\
& \quad \quad \quad (\text{dom } \mathcal{S}_{sb} \cup \mathcal{R} - \mathcal{L}) \ \mathcal{R}_j \\
& \quad \quad \quad = \text{release (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j)} \\
& \quad \quad \quad (\text{dom } \mathcal{S}_{sb}) \ \mathcal{R}_j \\
\text{proof} & \quad \{ \\
\text{fix} & \quad a \\
\text{assume} & \quad \text{a-in: } a \in \text{all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb}_j) \\
\text{have} & \quad (a \in (\text{dom } \mathcal{S}_{sb} \cup \mathcal{R} - \mathcal{L})) = (a \in \text{dom } \mathcal{S}_{sb}) \\
\text{proof} & \quad \{ \\
\text{from} & \quad \text{ownership-distinct \[\text{OF i-bound j-bound neq ts}_{sb}-i \ jth\]} \\
\text{have} & \quad \text{A-dist: } A \cap (\mathcal{O}_j \cup \text{all-acquired sb}_j) = \{\} \\
\text{by} & \quad (\text{auto simp add: sb Ghost}_{sb}) \\
\text{from} & \quad \text{all-shared-acquired-or-owned \[\text{OF sharing-consis \[\text{OF j-bound jth\]}\] a-in} \\
& \quad \text{all-shared-append \[\text{of (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb}_j) \\
& \quad \quad \quad \text{(dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb}_j)]} \\
\text{have} & \quad \text{a-in: } a \in \mathcal{O}_j \cup \text{all-acquired sb}_j \\
\text{by} & \quad \text{auto} \\
\text{with} & \quad \text{ownership-distinct \[\text{OF i-bound j-bound neq ts}_{sb}-i \ jth\]} \\
\text{have} & \quad a \notin (\mathcal{O}_{sb} \cup \text{all-acquired sb}) \quad \text{by} \quad \text{auto} \\
\text{with} & \quad \text{A-dist R-owned A-R A-shared-owned L-subset a-in} \\
\text{obtain} & \quad a \notin R \quad \text{and} \quad a \notin L \\
\text{by} & \quad \text{fastforce} \\
\text{then show} & \quad \text{thesis} \quad \text{by} \quad \text{auto} \\
\text{qed} & \quad \text{by}\end{align*}
then
show \(?\)thesis
apply -
apply (rule release-all-shared-exchange)
apply auto
done
qed

note release-commute = this
from ownership-distinct-axioms have ownership-distinct ts_{sb}.
from sharing-consis-axioms have sharing-consis S_{sb} ts_{sb}.

have (ts_{sb} \[ i := (p_{sb}, is_{sb}, \emptyset_{sb}, \emptyset, D_{sb}, O_{sb} \cup A - R, augment-rels (dom S_{sb}) R \]
\[ R_{sb} \], m_{sb}, S_{sb} \']) \sim (ts, m, S)
apply (rule sim-config, intros)
apply (simp add: m flush-commute)
apply (clarsimp simp add: S S' share-commute)
using leq
apply simp
using i-bound i-bound' ts-sim ts-i is-sim D
apply (clarsimp simp add: Let-def nth-list-update sb suspends Ghost_{sb} \[ R_{sb} \] \[ S_{sb} \] \[ S_{sb}' \] share-commute)
apply clarsimp
apply auto
done
ultimately
show \(?\)thesis
using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct'
valid-dd' valid-sops' load-tmps-fresh' enough-flushes'
valid-program-history' valid'
m_{sb}' S_{sb}' ts_{sb}'
by (auto simp del: fun-upd-apply simp add: O_{sb} \[ R_{sb} \] )
qed

next
case (Program i p_{sb} is_{sb} \emptyset_{sb} D_{sb} O_{sb} R_{sb} p_{sb}' mis)
then obtain
\[ ts_{sb}' \colon ts_{sb}' = ts_{sb}[i := (p_{sb}', is_{sb}@mis, \emptyset_{sb}, \emptyset)@[Prog_{sb} p_{sb} p_{sb}' mis], D_{sb}, O_{sb}, R_{sb}]\]
and
i-bound: i < length ts_{sb} and
\( ts_{sb}' \colon ts_{sb}' \[ i := (p_{sb}', is_{sb}@mis, \emptyset_{sb}, D_{sb}, O_{sb}, R_{sb}) \) and
\( prog' \colon \emptyset_{sb} \vdash p_{sb} \rightarrow_{p} (p_{sb}' , mis) \) and
\( S_{sb}' : S_{sb}' = S_{sb} \) and

682
\[ m_{sb}: m_{sb}' = m_{sb} \]

by auto

from sim obtain
m: m = flush-all-until-volatile-write \( t_{sb} \) \( m_{sb} \)
and
\( S: S = share-all-until-volatile-write \) \( t_{sb} \) \( S_{sb} \)
and
leq: length \( t_{sb} \) = length \( ts \)
and
\( \text{ts-sim: } \forall i < \text{length } t_{sb}, \)

let (p, is_{sb}, \( \emptyset \), sb, \( D_{sb}, O_{sb}, \mathcal{R} \)) = \( t_{sb} \) ! i;

suspends = dropWhile (Not \( \circ \) is-volatile-Write \( t_{sb} \) sb)
in \( \exists is \ D, \) instrs suspends @ is_{sb} = is @ prog-instrs suspends \( \land \)
\[ D_{sb} = (D \lor \text{outstanding-refs is-volatile-Write}_{sb} \) sb \( \neq \{\}) \) \( \land \)
\[ t_{s} ! i = \]

(hd-prog p suspends,

is,

\( \emptyset \xrightarrow{i} (\text{dom } \emptyset - \text{read-tmps suspends}), (), \)

\( D, \)

acquired True (takeWhile (Not \( \circ \) is-volatile-Write \( t_{sb} \) sb) \( O_{sb}, \)

release (takeWhile (Not \( \circ \) is-volatile-Write \( t_{sb} \) sb) (dom \( S_{sb} \) ) \( \mathcal{R} \))

by cases blast

from i-bound leq have i-bound': i < length ts
by auto

have split-sb: sb = takeWhile (Not \( \circ \) is-volatile-Write \( t_{sb} \) sb @ dropWhile (Not \( \circ \)

is-volatile-Write \( t_{sb} \) sb)

(is sb = ?take-sb@?drop-sb)

by simp

from ts-sim [rule-format, OF i-bound] \( t_{sb} \) obtain suspends is \( D \) where
suspends: suspends = dropWhile (Not \( \circ \) is-volatile-Write \( t_{sb} \) sb)
and
is-sim: instrs suspends @ is_{sb} = is @ prog-instrs suspends and
\( D: D_{sb} = (D \lor \text{outstanding-refs is-volatile-Write}_{sb} \) sb \( \neq \{\}) \) \( \land \)
\[ t_{s} ! i = \]

(hd-prog p_{sb} suspends, is,

\( \emptyset \xrightarrow{i} (\text{dom } \emptyset - \text{read-tmps suspends}), (), \)

\( D, \)

acquired True ?take-sb \( O_{sb}, \)

release ?take-sb (dom \( S_{sb} \) ) \( \mathcal{R}_{sb} \)

by (auto simp add: Let-def)

from prog-step-preserve-valid [OF i-bound \( t_{sb} \) ! prog valid]

have valid': valid \( t_{sb}' \)

by (simp add: \( t_{sb}' \))

have valid-own': valid-ownership \( S_{sb}' \) \( t_{sb}' \)

proof (intro-locales)

show outstanding-non-volatile-refs-owned-or-read-only \( S_{sb}' \) \( t_{sb}' \)

proof

from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound \( t_{sb} \) !]

have non-volatile-owned-or-read-only False \( S_{sb} \) \( O_{sb} \) (sb@[Prog_{sb} P_{sb} P_{sb}' mis])

by (auto simp add: non-volatile-owned-or-read-only-append)

683
from outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this]
show ?thesis by (simp add: ts\sb′ S\sb′)
qed

next
show outstanding-volatile-writes-unowned-by-others ts\sb′
proof

have out: outstanding-refs is-volatile-Write\sb (sb@[Prog\sb p\sb p\sb′ mis]) \subseteq
outstanding-refs is-volatile-Write\sb sb
by (auto simp add: outstanding-refs-conv )
from outstanding-volatile-writes-unowned-by-others-store-buffer
[OF i-bound ts\sb′-i this]
show ?thesis by (simp add: ts\sb′ all-acquired-append)
qed

next
show read-only-reads-unowned ts\sb′
proof

have ro: read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write\sb)
(sb@[Prog\sb p\sb p\sb′ mis]))
(dropWhile (Not \circ is-volatile-Write\sb) (sb@[Prog\sb p\sb p\sb′ mis]))
\subseteq read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write\sb) sb) O\sb)
(dropWhile (Not \circ is-volatile-Write\sb) sb)
apply (case-tac outstanding-refs (is-volatile-Write\sb) sb = {})
apply (simp-all add: outstanding-vol-write-take-drop-append
all-acquired-append read-only-reads-append )
done

have O\sb \cup all-acquired (sb@[Prog\sb p\sb p\sb′ mis]) \subseteq O\sb \cup all-acquired sb
by (auto simp add: all-acquired-append)
from read-only-reads-unowned-nth-update [OF i-bound ts\sb′-i ro this]
show ?thesis
by (simp add: ts\sb′)
qed

qed

from valid-last-prog [OF i-bound ts\sb′-i]
have last-prog: last-prog p\sb sb = p\sb.

have valid-hist′; valid-history program-step ts\sb′
proof

from valid-history [OF i-bound ts\sb′-i]
have history-consistent v\sb (hd-prog p\sb sb) sb.
from history-consistent-append-Prog\sb [OF prog this last-prog]
have hist-consis′; history-consistent v\sb (hd-prog p\sb′ (sb@[Prog\sb p\sb p\sb′ mis]))
(sb@[Prog\sb p\sb p\sb′ mis]).
from valid-history-nth-update [OF i-bound this]
show thesis by (simp add: ts_{sb}')
qed

have valid-reads' : valid-reads m_{sb} ts_{sb}'
proof
  from valid-reads [OF i-bound ts_{sb}-i]
  have reads-consistent False O_{sb} m_{sb} sb .
  from reads-consistent-snoc-Prog_{sb} [OF this]
  have reads-consistent False O_{sb} m_{sb} (sb@[Prog_{sb} p_{sb} p_{sb}' mis]).
  from valid-reads-nth-update [OF i-bound this]
  show thesis by (simp add: ts_{sb}')
qed

have valid-sharing' : valid-sharing S_{sb}' ts_{sb}'
proof
  intro-locale
  from outstanding-non-volatile-writes-unshared [OF i-bound ts_{sb}-i]
  have non-volatile-writes-unshared S_{sb} (sb@[Prog_{sb} p_{sb} p_{sb}' mis])
by (auto simp add: non-volatile-writes-unshared-append)
  from outstanding-non-volatile-writes-unshared-nth-update [OF i-bound this]
  show outstanding-non-volatile-writes-unshared S_{sb}' ts_{sb}'
by (simp add: ts_{sb}' S_{sb}')
next
  from sharing-consis [OF i-bound ts_{sb}-i]
  have sharing-consistent S_{sb} O_{sb} (sb@[Prog_{sb} p_{sb} p_{sb}' mis])
by (auto simp add: sharing-consistent-append)
  from sharing-consis-nth-update [OF i-bound this]
  show sharing-consis S_{sb}' ts_{sb}'
by (simp add: ts_{sb}' S_{sb}')
next
  from read-only-unowned-nth-update [OF i-bound read-only-unowned [OF i-bound
ts_{sb}-i] ]
  show read-only-unowned S_{sb}' ts_{sb}'
by (simp add: S_{sb}' ts_{sb}')
next
  from unowned-shared-nth-update [OF i-bound ts_{sb}-i subset-refl]
  show unowned-shared S_{sb}' ts_{sb}'
by (simp add: ts_{sb}' S_{sb}')
next
  from no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb}-i]

  have no-write-to-read-only-memory S_{sb} (sb @[Prog_{sb} p_{sb} p_{sb}' mis])
by (simp add: no-write-to-read-only-memory-append)

  from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
  show no-outstanding-write-to-read-only-memory S_{sb}' ts_{sb}'
by (simp add: S_{sb}' ts_{sb}')
qed
have tmps-distinct': tmps-distinct ts_{sb}'
proof (intro-locale)
  from load-tmps-distinct [OF i-bound ts_{sb}-i]
  have distinct-load-tmps is_{sb}.
  with distinct-load-tmps-prog-step [OF i-bound ts_{sb}-i prog valid]
  have distinct-load-tmps (is_{sb}@mis)
by (auto simp add: distinct-load-tmps-append)

  from load-tmps-distinct-nth-update [OF i-bound this]
  show load-tmps-distinct ts_{sb}'
by (simp add: ts_{sb}')
next
  from read-tmps-distinct [OF i-bound ts_{sb}-i]
  have distinct-read-tmps (sb@[Prog_{sb} p_{sb} p_{sb}' mis])
by (simp add: distinct-read-tmps-append)
  from read-tmps-distinct-nth-update [OF i-bound this]
  show read-tmps-distinct ts_{sb}'
by (simp add: ts_{sb}')
qed

have valid-dd': valid-data-dependency ts_{sb}'
proof –
  from data-dependency-consistent-instrs [OF i-bound ts_{sb}-i]
  have data-dependency-consistent-instrs (dom θ_{sb}) is_{sb}.
  with valid-data-dependency-prog-step [OF i-bound ts_{sb}-i prog valid]
load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
  obtain
data-dependency-consistent-instrs (dom θ_{sb}) (is_{sb}@mis)
load-tmps (is_{sb}@mis) ∪ (fst ' write-sops (sb@[Prog_{sb} p_{sb} p_{sb}' mis])) = {}
by (force simp add: load-tmps-append data-dependency-consistent-instrs-append
  write-sops-append)
  from valid-data-dependency-nth-update [OF i-bound this]
  show ?thesis
by (simp add: ts_{sb}')
qed

have load-tmps-fresh': load-tmps-fresh ts_{sb}'
proof –
  from load-tmps-fresh [OF i-bound ts_{sb}-i]
load-tmps-fresh-prog-step [OF i-bound ts_{sb}-i prog valid]
  have load-tmps (is_{sb}@mis) ∩ dom θ_{sb} = {}
by (auto simp add: load-tmps-append)
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis
by (simp add: ts sb')
qed

have enough-flushs': enough-flushs ts sb'
proof
from clean-no-outstanding-volatile-Write sb [OF i-bound ts sb'-i]
have ¬D sb → outstanding-refs is-volatile-Write sb (sb@[Prog sb p sb p sb' mis]) = {}
by (auto simp add: outstanding-refs-append)

from enough-flushs-nth-update [OF i-bound this]
show ?thesis
by (simp add: ts sb')
qed

have valid-sops': valid-sops ts sb'
proof
from valid-store-sops [OF i-bound ts sb'-i] valid-sops-prog-step [OF prog]
valid-implies-valid-prog[OF i-bound ts sb'-i valid]
have valid-store: ∀sop∈store-sops (is sb @mis). valid-sop sop
by (auto simp add: store-sops-append)

from valid-write-sops [OF i-bound ts sb'-i]
have ∀sop∈write-sops (sb@[Prog sb p sb p sb' mis]). valid-sop sop
by (auto simp add: write-sops-append)

from valid-sops-nth-update [OF i-bound this valid-store]
show ?thesis
by (simp add: ts sb')
qed

have valid-program-history':valid-program-history ts sb'
proof
from valid-program-history [OF i-bound ts sb'-i]
have causal-program-history is sb'.
from causal-program-history-Prog sb [OF this]
have causal': causal-program-history (is sb@mis) (sb@[Prog sb p sb p sb' mis]).
from last-prog-append-Prog sb
have last-prog p sb' (sb@[Prog sb p sb p sb' mis]) = p sb'.
from valid-program-history-nth-update [OF i-bound causal' this]
show ?thesis
by (simp add: ts sb')
qed

show ?thesis
proof (cases outstanding-refs is-volatile-Write sb sb = {})
case True
from True have flush-all: takeWhile (Not ∘ is-volatile-Write sb) sb = sb
by (auto simp add: outstanding-refs-conv)
from True have suspend-nothing: dropWhile (Not o is-volatile-Write sb) sb = []
by (auto simp add: outstanding-refs-conv)

hence suspends-empty: suspends = []
by (simp add: suspends)

from suspends-empty is-sim have is: is = is sb
by (simp)

from ts-i have ts-i: ts ! i = (p sb, is sb, θ sb, ()
D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb)
by (simp add: suspends-empty is)

from direct-computation.Program [OF i-bound ts-i prog]
have (ts, m, S) ⇒ d (ts[i := (p sb′, is sb @ mis, θ sb, ()
D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb]), m, S).

moreover

note flush-commute = flush-all-until-volatile-write-append-Prog sb-commute [OF i-bound ts sb-i]

from True
have suspend-nothing′:
(dropWhile (Not o is-volatile-Write sb) (sb @ [Prog sb p sb p sb′ mis])) = []
by (auto simp add: outstanding-refs-conv)

note share-commute =
share-all-until-volatile-write-update-sb [OF share-append-Prog sb i-bound ts sb-i]

from D
have D′: D sb = (D ∨ outstanding-refs is-volatile-Write sb (sb@[Prog sb p sb p sb′ mis])
≠ {})
by (auto simp: outstanding-refs-append)

have (ts sb [i := (p sb′, is sb @ mis, θ sb, sb@[Prog sb p sb p sb′ mis]], D sb, O sb, R sb]),
(Leb sb, S sb′) ∼
(ts[i := (p sb′, is sb @ mis, θ sb, ()), D,
acquired True (takeWhile (Not o is-volatile-Write sb)
(sb@[Prog sb p sb p sb′ mis])) O sb,
release (sb@[Prog sb p sb p sb′ mis]) (dom S sb) R sb], m, S)
apply (rule sim-config.intros)
apply (simp add: m flush-commute)
apply (clarsimp simp add: S S sb′ share-commute)
using leq
apply simp
using i-bound i-bound′ ts-sim ts-i D′
apply (clarsimp simp add: Let-def nth-list-update flush-all suspend-nothing′ Prog sb S sb′

688
ultimately show thesis
using valid-own’ valid-list’ valid-reads’ valid-shares’ tmps-distinct’ m神州’
valid-dd’ valid-sops’ load-tmps-fresh’ enough-flushes’ valid-shares’
valid-program-history’ valid’
S神州’ t神州’
by (auto simp del: fun-upd-apply simp add: acquired-append-神州神州 release-append-神州神州 release-append flush-all)

next

then obtain r where r-in: r ∈ set神州 and volatile-r: is-volatile-Write神州神州 r
by (auto simp add: outstanding-refs-conv)
from takeWhile-dropWhile-real-prefix
[OF r-in, OF (Not ◦ is-volatile-Write神州神州), simplified, OF volatile-r]
obtain a’ v’ sb’ / sop’ A’ L’ R’ W’ where
sb-split: sb = takeWhile (Not ◦ is-volatile-Write神州神州) sb @ Write神州神州 True a’ sop’ v’ A’ L’ R’
W’ # sb’
and
drop: dropWhile (Not ◦ is-volatile-Write神州神州) sb = Write神州神州 True a’ sop’ v’ A’ L’ R’ W’ #
sb’
apply (auto)
subgoal for y
apply (case-tac y)
apply auto
done
done
from drop suspends have suspends’: suspends = Write神州神州 True a’ sop’ v’ A’ L’ R’
W’ # sb’
by simp

have (ts, m, S) ⇒ₜₜ’ (ts, m, S) by auto

moreover

note flush-commute= flush-all-until-volatile-write-append-神州神州-commute [OF i-bound t神州神州-i]

have Write神州神州 True a’ sop’ v’ A’ L’ R’ W’ ∈ set神州
by (subst神州神州-split) auto

from dropWhile-append1 [OF this, OF (Not ◦ is-volatile-Write神州神州)]
have drop-app-comm:
(dropWhile (Not ◦ is-volatile-Write神州神州) (sb @ [神州神州 p神州神州 p神州神州’ mis])) =
dropWhile (Not ◦ is-volatile-Write神州神州) sb @ [神州神州 p神州神州 p神州神州’ mis]
by simp

note share-commute =
share-all-until-volatile-write-update-sb [OF share-append-Progsb i-bound ts_{sb}\cdot i]

from \mathcal{D}

have \mathcal{D}' : \mathcal{D}_{sb} = (\mathcal{D} \lor \text{outstanding-refs is-volatile-Write}_{sb sb}(sb\cdot p_{sb} p_{sb}' \cdot mis))
\neq \{\}

by (auto simp: outstanding-refs-append)

have (ts_{sb} [i := (p_{sb}', i_{sb}\cdot 0@mis, i_{sb}, sb\cdot [Progsb p_{sb} p_{sb}' \cdot mis], \mathcal{D}_{sb}, \mathcal{O}_{sb}, \mathcal{R}_{sb}]),
\quad m_{sb}, S_{sb}') 
\sim 
(t, s, m, S)

apply (rule sim-config.intros)

apply (simp add: m flush-commute)

apply (clarsimp simp add: \mathcal{S}_{sb}' share-commute)

using leq

apply simp

using i-bound i-bound' ts-sim ts-i is-sim_suspend suspend' [simplified suspend'] \mathcal{D}'

apply (clarsimp simp add: Progsb instrs-append prog-instrs-append read-tmps-append hd-prog-append-Progsb
\quad acquired-append-Progsb release-append-Progsb release-append \mathcal{S}_{sb}'
\quad split: if-split-asm)

done

ultimately show ?thesis

using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' m_{sb}'
\quad valid-dd' valid-sops' load-tmps-fresh' enough-flushes' valid-sharing'
\quad valid-program-history' valid'
\quad \mathcal{S}_{sb}' t_{sb}'

by (auto simp del: fun-upd-apply)

qed

qed

qed

theorem (in xvalid-program) concurrent-direct-steps-simulates-store-buffer-history-steps:

assumes step-sb: \((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sb} (ts_{sb}', m_{sb}', S_{sb}')\)

assumes valid-own: valid-ownership S_{sb} ts_{sb}

assumes valid-sb-reads: valid-reads m_{sb} ts_{sb}

assumes valid-hist: valid-history program-step ts_{sb}

assumes valid-sharing: valid-sharing S_{sb} ts_{sb}

assumes tmps-distinct: tmps-distinct ts_{sb}

assumes valid-sops: valid-sops ts_{sb}

assumes valid-dd: valid-data-dependency ts_{sb}

assumes load-tmps-fresh: load-tmps-fresh ts_{sb}

assumes enough-flushs: enough-flushs ts_{sb}

assumes valid-program-history: valid-program-history ts_{sb}

assumes valid: valid ts_{sb}

shows \(ts S m. (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S) \Rightarrow \text{safe-reach-direct} \text{ safe-delayed} (ts, m, S) \Rightarrow \)
valid-ownership $S_{sb}$' $t_{sb}$' $\land$ valid-reads $m_{sb}$' $t_{sb}$' $\land$ valid-history program-step $t_{sb}$'

$\land$

valid-sharing $S_{sb}$' $t_{sb}$' $\land$ tmtps-distinct $t_{sb}$' $\land$ valid-data-dependency $t_{sb}$' $\land$

valid-sops $t_{sb}$' $\land$ load-tmtps-fresh $t_{sb}$' $\land$ enough-flushs $t_{sb}$' $\land$

valid-program-history $t_{sb}$' $\land$ valid $t_{sb}$' $\land$

($\exists t'$ $m'$ $S'$. $(t', m', S)$ $\Rightarrow_d$' $(t', m', S')$' $\land$ $(t_{sb}', m_{sb}', S_{sb}')$' $\sim$ $(t', m', S')$)

using step-sb valid-own valid-sb-reads valid-hist valid-sharing tmtps-distinct valid-sops valid-dd load-tmtps-fresh enough-flushs valid-program-history valid

proof (induct rule: converse-rtranclp-induct-sbh-steps)

case refl thus \text{case} by auto

case \text{case} (step $t_{sb}$ $m_{sb}$ $S_{sb}$ $t_{sb}$'' $m_{sb}'' S_{sb}''$)

\text{note} first = ($t_{sb}$, $m_{sb}$, $S_{sb}$) $\Rightarrow_{sbh}$ ($t_{sb}$'', $m_{sb}''$, $S_{sb}''$):

\text{note} sim = ($t_{sb}$, $m_{sb}$, $S_{sb}$) $\sim$ ($t$, $m$, $S$):

\text{note} safe-reach = \text{safe-reach-direct safe-delayed ($t$, $m$, $S$)}:

\text{note} valid-own = \text{valid-ownership $S_{sb}$ $t_{sb}$}:

\text{note} valid-reads = \text{valid-reads $m_{sb}$ $t_{sb}$}:

\text{note} valid-sharing = \text{valid-sharing $S_{sb}$ $t_{sb}$}:

\text{note} tmtps-distinct = \text{tmtps-distinct $t_{sb}$}:

\text{note} valid-sops = \text{valid-sops $t_{sb}$}:

\text{note} valid-dd = \text{valid-data-dependency $t_{sb}$}:

\text{note} load-tmtps-fresh = \text{load-tmtps-fresh $t_{sb}$}:

\text{note} enough-flushs = \text{enough-flushs $t_{sb}$}:

\text{note} valid-prog-hist = \text{valid-program-history $t_{sb}$}:

\text{note} valid = \text{valid $t_{sb}$}:

from concurrent-direct-steps-simulates-store-buffer-history-step [OF first valid-own valid-reads valid-hist valid-sharing tmtps-distinct valid-sops valid-dd load-tmtps-fresh enough-flushs valid-prog-hist valid sim safe-reach]

obtain $t$'' $m$'' $S$'' \text{where}

valid-own'': \text{valid-ownership $S_{sb}$'' $t_{sb}$'' and}

valid-reads'': \text{valid-reads $m_{sb}$'' $t_{sb}$'' and}

valid-hist'': \text{valid-history program-step $t_{sb}$'' and}

valid-sharing'': \text{valid-sharing $S_{sb}$'' $t_{sb}$'' and}

tmps-dist'': \text{tmpps-distinct $t_{sb}$'' and}

valid-dd'': \text{valid-data-dependency $t_{sb}$'' and}

valid-sops'': \text{valid-sops $t_{sb}$'' and}

load-tmtps-fresh'': \text{load-tmtps-fresh $t_{sb}$'' and}

enough-flushs'': \text{enough-flushs $t_{sb}$'' and}

valid-prog-hist'': \text{valid-program-history $t_{sb}$'' and}

valid'': \text{valid $t_{sb}$'' and}

steps: ($t$, $m$, $S$) $\Rightarrow_d$ ($t$'', $m$'', $S$'') and\n
sim: ($t_{sb}$'', $m_{sb}''$, $S_{sb}''$) $\sim$ ($t''$, $m''$, $S''$)

by blast

from step.hyps (3) [OF sim safe-reach-steps [OF safe-reach steps] valid-own'' valid-reads'' valid-hist'' valid-sharing'']

691
obtain \( ts' \in S' \) where
valid: valid-ownership \( S_{sb} \) \( ts_{sb} \) \( ts_{sb}' \) \( ts_{sb}' \) valid-reads \( m_{sb} \) \( ts_{sb} \) \( ts_{sb}' \) valid-history program-step \( ts_{sb} \)
valid-sharing \( S_{sb} \) \( ts_{sb} \) \( ts_{sb}' \) \( ts_{sb}' \) valid-data-dependency \( ts_{sb} \)
valid-sops \( ts_{sb} \) \( ts_{sb}' \) \( ts_{sb}' \) load-tmps-fresh \( ts_{sb} \) \( ts_{sb}' \) \( ts_{sb}' \) enough-flushes \( ts_{sb} \)
valid-program-history \( ts_{sb} \) \( ts_{sb}' \) \( ts_{sb}' \) valid \( ts_{sb}' \) \( ts_{sb}' \) and

last: \( ts'', m'', S'' \) \( \Rightarrow_d^* (ts', m', S') \) \( \and \)
sim': \( (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S') \)

by blast

note steps also note last
finally show \( ?\) case
using valid sim'
by blast
qed

sublocale initial_{sb} \subseteq tmps-distinct ..
locale xvalid-program-progress = program-progress + xvalid-program

theorem (in xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-history-execution:
assumes exec-sb: \( (ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbh}^* (ts_{sb}', m_{sb}', S_{sb}') \)
assumes init: initial_{sb} \( ts_{sb} \) \( S_{sb} \)
assumes valid: valid \( ts_{sb} \)
assumes sim: \( (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S) \)
assumes safe: safe-reach-direct safe-free-flowing \( (ts, m, S) \)
shows \( \exists ts' \in S', (ts, m, S) \Rightarrow_d^* (ts', m', S') \) \( \land \)
\( (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S') \)

proof --
from init interpret ini: initial_{sb} \( ts_{sb} \) \( S_{sb} \) .
from safe-free-flowing-implies-safe-delayed' [OF init sim safe]
have safe-delayed: safe-reach-direct safe-delayed \( (ts, m, S) \).
from local.ini.valid-ownership-axioms have valid-ownership \( S_{sb} \) \( ts_{sb} \) .
from local.ini.valid-reads-axioms have valid-reads \( m_{sb} \) \( ts_{sb} \).
from local.ini.valid-history-axioms have valid-history program-step \( ts_{sb} \).
from local.ini.valid-sharing-axioms have valid-sharing \( S_{sb} \) \( ts_{sb} \).
from local.ini.tmps-distinct-axioms have tmps-distinct \( ts_{sb} \).
from local.ini.valid-sops-axioms have valid-sops \( ts_{sb} \).
from local.ini.valid-data-dependency-axioms have valid-data-dependency \( ts_{sb} \).
from local.ini.load-tmps-fresh-axioms have load-tmps-fresh \( ts_{sb} \).
from local.ini.enough-flushs-axioms have enough-flushs \( ts_{sb} \).
from local.ini.valid-program-history-axioms have valid-program-history \( ts_{sb} \).
from concurrent-direct-steps-simulates-store-buffer-history-steps [OF exec-sb
(\( valid-ownership \) \( S_{sb} \) \( ts_{sb} \))
(\( valid-reads \) \( m_{sb} \) \( ts_{sb} \)) \( valid-history program-step \) \( ts_{sb} \))
(\( valid-sharing \) \( S_{sb} \) \( ts_{sb} \)) \( tmps-distinct \) \( ts_{sb} \) \( valid-sops \) \( ts_{sb} \))
(\( valid-data-dependency \) \( ts_{sb} \) \( load-tmps-fresh \) \( ts_{sb} \)) \( enough-flushs \) \( ts_{sb} \))
(\( valid-program-history \) \( ts_{sb} \)) \( valid sim safe-delayed \)

692
lemma filter-is-Write\(_{sb}\)-Cons-Write\(_{sb}\): filter is-Write\(_{sb}\) \(xs = Write\(_{sb}\) volatile a sop v A L R W#ys\)
\[\implies \exists rs rws. (\forall r \in set rs. is-Read\(_{sb}\) r \lor is-Prog\(_{sb}\) r \lor is-Ghost\(_{sb}\) r) \land xs=rs@Write\(_{sb}\) volatile a sop v A L R W#rws \land ys=filter is-Write\(_{sb}\) rws\]

proof (induct \(xs\))
  case Nil thus ?case by simp
next
  case (Cons \(x\) \(xs\))
  note feq = \(\langle filter is-Write\(_{sb}\) (x#xs) = Write\(_{sb}\) volatile a sop v A L R W#ys \rangle\)

  show ?case
  proof (cases \(x\))
    case (Write\(_{sb}\) volatile\(\)\(\) a\(\)\(\) t\(\)\(\) v\(\)\(\) #rs)
    from feq have filter is-Write\(_{sb}\) (x#xs) = Write\(_{sb}\) volatile a sop v A L R W#ys

    by (simp add: Write\(_{sb}\))
    done

  next
  from feq have filter is-Write\(_{sb}\) (x#xs) = Write\(_{sb}\) volatile a sop v A L R W#ys

    by (simp add: Read\(_{sb}\))
    from Cons.hyps [OF this] obtain rs rws where
    \(\forall r \in set rs. is-Read\(_{sb}\) r \lor is-Prog\(_{sb}\) r \lor is-Ghost\(_{sb}\) r \land xs=rs@Write\(_{sb}\) volatile a sop v A L R W#rws \land ys=filter is-Write\(_{sb}\) rws\)

    by clarsimp
    then show ?thesis
      apply (rule-tac x=Read\(_{sb}\) t v #rs in exI)
    done

next
  case (Prog\(_{sb}\) p\(_1\) p\(_2\) mis)
  from feq have filter is-Write\(_{sb}\) (x#xs) = Write\(_{sb}\) volatile a sop v A L R W#ys

    by (simp add: Prog\(_{sb}\))
    from Cons.hyps [OF this] obtain rs rws where
    \(\forall r \in set rs. is-Read\(_{sb}\) r \lor is-Prog\(_{sb}\) r \lor is-Ghost\(_{sb}\) r \land xs=rs@Write\(_{sb}\) volatile a sop v A L R W#rws \land ys=filter is-Write\(_{sb}\) rws\)

  qed
ys=filter is-Write\(sb\) rws
by clarsimp
then show \(\lnot\)thesis
  apply-
  apply (rule-tac x=Prog\(sb\) p1 p2 mis\#rs in exI)
  apply (rule-tac x=rws in exI)
  apply (simp add: Prog\(sb\))
  done
next
case (Ghost\(sb\) A’ L’ R’ W’)
from feq have filter is-Write\(sb\) xs = Write\(sb\) volatile a sop v A L R W # ys
  by (simp add: Ghost\(sb\))
from Cons.hyps [OF this] obtain rs rws where
  \(\forall r \in\) set rs. is-Read\(sb\) r \(\vee\) is-Prog\(sb\) r \(\vee\) is-Ghost\(sb\) r
  and xs=rs @ Write\(sb\) volatile a sop v A L R W# rws
  and ys=filter is-Write\(sb\) rws
by clarsimp
then show \(\lnot\)thesis
  apply-
  apply (rule-tac x=Ghost\(sb\) A’ L’ R’ W’#rs in exI)
  apply (rule-tac x=rws in exI)
  apply (simp add: Ghost\(sb\))
  done
qed
qed

lemma filter-is-Write\(sb\)-empty: filter is-Write\(sb\) xs = []
  \(\Rightarrow\) (\(\forall r \in\) set xs. is-Read\(sb\) r \(\vee\) is-Prog\(sb\) r \(\vee\) is-Ghost\(sb\) r)
proof (induct xs)
case Nil thus ?case by simp
next
case (Cons x xs)
note feq = (filter is-Write\(sb\) (x#xs) = []);
show ?case
proof (cases x)
case (Write\(sb\) volatile’ a’ v’)
  with feq have False
  by simp
  thus ?thesis ..
next
case (Read\(sb\) a’ v’)
from feq have filter is-Write\(sb\) xs = []
  by (simp add: Read\(sb\))
from Cons.hyps [OF this] obtain
  \(\forall r \in\) set xs. is-Read\(sb\) r \(\vee\) is-Prog\(sb\) r \(\vee\) is-Ghost\(sb\) r
  by clarsimp
then show ?thesis
  by (simp add: Read\(sb\))
next
case (Prog\(sb\) p2 p2 mis)
from feq have filter is-Write \(sb\) \(xs = []\)
  by (simp add: Prog \(sb\))
from Cons.hyps [OF this] obtain
  \(\forall r \in \text{set } xs. \text{is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r\)
  by clarsimp
then show \(?\)thesis
  by (simp add: Prog \(sb\))
next
case (\(\text{Ghost}_{sb} A' L' R' W'\))
from feq have filter is-Write \(sb\) \(xs = []\)
  by (simp add: Ghost \(sb\))
from Cons.hyps [OF this] obtain
  \(\forall r \in \text{set } xs. \text{is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r\)
  by clarsimp
then show \(?\)thesis
  by (simp add: Ghost \(sb\))
qed

qed

lemma flush-reads-program: \(\bigwedge OSR.\)
  \(\forall r \in \text{set } sb. \text{is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r \implies\)
  \(\exists O'R' S'. (m, sb, O', R', S') \rightarrow_{\ell} (m, [], O'R', S')\)
proof (induct sb)
case Nil thus \(?\)case by auto
next
case (Cons \(x \# sb\))
ote \(\forall r \in \text{set } (x \# sb). \text{is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r\)
then obtain \(x: \text{is-Read}_{sb} x \lor \text{is-Prog}_{sb} x \lor \text{is-Ghost}_{sb} x\) \(\text{and } sb: \forall r \in \text{set } sb. \text{is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r\)
  by (cases \(x\)) auto

\{
  assume is-Read_{sb} \(x\)
  then obtain volatile \(a \lor t\) \(v\) where \(x: x=\text{Read}_{sb} \text{volatile } a \land t\lor \text{volatile } a \land t\)
    by (cases \(x\)) auto

  have \((m, \text{Read}_{sb} \text{volatile } a \land t \# sb, O', R', S') \rightarrow_{\ell} (m, sb, O', R', S')\)
    by (rule Read_{sb})
  also
  from Cons.hyps [OF \(sb\)] obtain \(O'S' acq' R'\)
    \(\text{where } (m, sb, O', R', S') \rightarrow_{\ell} (m, [], O', R', S')\) by blast
  finally
  have \(?\)case
    by (auto simp add: \(x\))
\}
moreover
\{
  assume is-Prog_{sb} \(x\)
  then obtain \(p_1 \land p_2 \land mis\) \(\text{where } x: x=\text{Prog}_{sb} p_1 \land p_2 \land mis\)
by (cases x) auto

have (m,Prog sb p1 p2 mis #sb,O,R,S) →f (m,sb,O,R,S)
  by (rule Prog sb)
also
from Cons.hyps [OF sb] obtain O' R' S' acq'
where (m, sb,O,R,S) →f* (m, [],O',R',S') by blast
finally
have ?case
  by (auto simp add: x)
}
moreover
{
  assume is-Ghost sb x
  then obtain A L R W where x = Ghost sb A L R W
  by (cases x) auto

  have (m,Ghost sb A L R W#sb,O,R,S) →f (m,sb,O ∪ A − R,augment-rels (dom S) R R,S ⊕ W R ⊖ A L)
    by (rule Ghost)
  also
  from Cons.hyps [OF sb] obtain O' S' R' acq'
  where (m, sb,O ∪ A − R ,augment-rels (dom S) R R,S ⊕ W R ⊖ A L) →f* (m, [],O',R',S') by blast
  finally
  have ?case
    by (auto simp add: x)
}
ultimately show ?case
  using x by blast
qed

lemma flush-progress: ∃ m' O' S' R'. (m,r#sb,O,R,S) →f (m',sb,O',R',S')
proof (cases r)
  case (Write sb volatile a sop v A L R W)
  from flush-step.Write sb [OF refl refl refl, of m volatile a sop v A L R W sb O R S]
  show ?thesis
    by (auto simp add: Write sb)
next
  case (Read sb volatile a t v)
  from flush-step.Read sb [of m volatile a t v sb O R S]
  show ?thesis
    by (auto simp add: Read sb)
next
  case (Prog sb p1 p2 mis)
  from flush-step.Prog sb [of m p1 p2 mis sb O R S]
  show ?thesis
    by (auto simp add: Prog sb)
next
case (Ghost哲学 A L R W)
from flush-step.Ghost [of m A L R W sb O R S]
show ?thesis
  by (auto simp add: Ghost)
qed

lemma flush-empty:
  assumes steps: (m, sb, O, R, S) →◦ (m', sb', O', R', S')
  shows sb=[] ==> m'=m ∧ sb'=[] ∧ O'=O ∧ R'=R ∧ S'=S
using steps
apply (induct rule: converse-rtranclp-induct5)
apply (auto elim: flush-step)
steps
proof
  using steps
next
  case (Ghost)
    assumes steps: (m, sb)
    shows sb="" = (if volatile then Map.empty else S)
    from hyps (3)
    have append-rest: (m", sb"@xs, O", R", S") →◦ (m', sb"@xs, O', R', S')
    from first
    show ?thesis
proof (cases)
  case (Write ab volatile A R W L a sop v)
  then obtain sb: sb=Write ab volatile a sop v A L R W#sb" and m": m"=m(a:=v)
and
O": O"=(if volatile then O ∪ A − R else O)
and
R": R"=(if volatile then Map.empty else R)
and
S": S"=(if volatile then S ⊕ W R ⊕ A L else S)
by auto
have (m,Write ab volatile a sop v A L R W#sb"@xs, O, R, S) →f
  (m(a:=v), sb"@xs, if volatile then O ∪ A − R else O, if volatile then Map.empty else R,
  if volatile then S ⊕ W R ⊕ A L else S)
apply (rule flush-step.Write)
apply auto
done
hence (m, sb"@xs, O, R, S) →f (m", sb"@xs, O", R", S")
by (simp add: sb m" O" R" S")
also note append-rest
finally show ?thesis .
next
  case (Read ab volatile a t v)
  then obtain sb: sb=Read ab volatile a t v #sb" and m": m"=m
and O": O"=O and S": S"=S and R": R"=R
by auto
have \((m, \text{Read}_{ab} \text{ volatile a t v}\#sb''@xs, O, \mathcal{R}, S) \rightarrow_f (m, sb''@xs, O, \mathcal{R}, S)\)
  by (rule flush-step, \text{Read}_{ab})
hence \((m, sb@xs, O, \mathcal{R}, S) \rightarrow_f (m'', sb''@xs, O'', \mathcal{R}'', S'')\)
  by (simp add: sb m'' O'' \mathcal{R}'' S'')
also note append-rest
finally show \(?thesis\).
next
case (Prog\_sb p1 p2 mis)
then obtain sb: sb = Prog\_sb p1 p2 mis # sb'' and m'': m'' = m
and \(O'': O'' = O \text{ and } S'': S'' = S \text{ and } \mathcal{R}'': \mathcal{R}'' = \mathcal{R}\)
  by auto
have \((m, \text{Prog}_{ab} p1 p2 \text{ mis # sb''@xs, O, } \mathcal{R}, S) \rightarrow_f (m, sb''@xs, O, \mathcal{R}, S)\)
  by (rule flush-step, Prog\_sb)
hence \((m, sb@xs, O, \mathcal{R}, S) \rightarrow_f (m'', sb''@xs, O'', \mathcal{R}'', S'')\)
  by (simp add: sb m'' O'' \mathcal{R}'' S'')
also note append-rest
finally show \(?thesis\).

next
case (Ghost A L R W)
then obtain sb: sb = Ghost\_sb A L R W # sb'' and m'': m'' = m
and \(O'': O'' = O \cup A - R \text{ and } S'': S'' = S \oplus_W R \ominus A L \text{ and } \mathcal{R}'': \mathcal{R}'' = \text{augment-rels (dom } S) R \mathcal{R}\)
  by auto
have \((m, \text{Ghost}_{ab} A L R W # sb''@xs, O, \mathcal{R}, S) \rightarrow_f (m, sb''@xs, O \cup A - R, \text{augment-rels (dom } S) R \mathcal{R}, S \oplus_W R \ominus A L)\)
  by (rule flush-step, Ghost)
hence \((m, sb@xs, O, \mathcal{R}, S) \rightarrow_f (m'', sb''@xs, O'', \mathcal{R}'', S'')\)
  by (simp add: sb m'' O'' \mathcal{R}'' S'')
also note append-rest
finally show \(?thesis\).
qed

lemmas store-buffer-step-induct =
store-buffer-step-induct [split-format (complete),
consumes 1, case-names SBWrite\_ab]

theorem flush-simulates-filter-writes:
assumes \(\text{step: } (m, sb, O, \mathcal{R}, S) \rightarrow_w (m', sb', \mathcal{O}', \mathcal{R}', S')\)
shows \(\bigwedge sb. O_h \mathcal{R}_h S_h. sb = \text{is-Write}_{ab} sb_h \Rightarrow \exists sb_h. \mathcal{O}_h \mathcal{R}_h S_h. (m, sb_h, O_h, \mathcal{R}_h, S_h) \rightarrow_f (m', sb_h', \mathcal{O}_h', \mathcal{R}_h', S_h') \land sb' = \text{is-Write}_{ab} sb'_h \land (sb' = [] \rightarrow sb_h = [])\)
using step
proof (induct rule: store-buffer-step-induct)
case (SBWrite\_ab m volatile a D f v A L R W sb \mathcal{O} \mathcal{R} S)
  note filter-Write\_sb = (Write\_ab volatile a (D, f) v A L R W# sb = filter is-Write\_sb sb_h)

  from filter-is-Write\_sb-Cons-Write\_sb [OF filter-Write\_sb [symmetric]]
obtain rs rws where 
rs-reads: \( \forall r \in \text{set } \text{rs} \quad \text{is-Read}_{sb} \ r \lor \text{is-Prog}_{sb} \ r \lor \text{is-Ghost}_{sb} \ r \) and
sb\_\text{h}: sb\_\text{h} = \text{rs} @ \text{Write}_{sb} \text{ volatile } a \ (D, f) \ v \ A \ L \ R \ W# \ rws and
sb: sb = \text{filter is-Write}_{sb} \ rws
by blast

from flush-reads-program [OF rs-reads] obtain \( O_{h}', \mathcal{R}_{h}', S_{h}' \) \text{acq}_{h}'
where (m, rs, \( O_{h}, \mathcal{R}_{h}, S_{h} \)) \rightarrow^{t^*} (m, [], \( O_{h}', \mathcal{R}_{h}', S_{h}' \)) by blast
from flush-append [OF this]
have (m, \text{Write}_{sb} \text{ volatile } a \ (D, f) \ v \ A \ L \ R \ W# \ rws, \( O_{h}, \mathcal{R}_{h}, S_{h} \)) \rightarrow^{t^*} (m, \text{Write}_{sb} \text{ volatile } a \ (D, f) \ v \ A \ L \ R \ W# \ rws, \( O_{h}', \mathcal{R}_{h}', S_{h}' \))
by simp
also
from flush-step.Write\_sb [OF refl refl refl, of m \text{volatile } a \ (D, f) \ v \ A \ L \ R \ W rws \( O_{h}', \mathcal{R}_{h}', S_{h}' \)]
obtain \( O_{h}''', \mathcal{R}_{h}''', S_{h}''' \) \text{acq}_{h}'''
where (m(a:=v), rws, \( O_{h}'', \mathcal{R}_{h}'', S_{h}'' \)) \rightarrow^{t} (m(a:=v), rws, \( O_{h}'', \mathcal{R}_{h}'', S_{h}'' \)) by blast
finally have steps: (m, sb\_h, \( O_{h}, \mathcal{R}_{h}, S_{h} \)) \rightarrow^{t^*} (m(a:=v), rws, \( O_{h}'', \mathcal{R}_{h}'', S_{h}''' \))
by (simp add: sb\_h sb)
show ?case
proof (cases sb)
  case Cons
  with steps sb show ?thesis
  by fastforce
next
  case Nil
from filter-is-Write\_sb-empty [OF sb [simplified Nil, symmetric]]
have \( \forall r \in \text{set } \text{rs} \quad \text{is-Read}_{sb} \ r \lor \text{is-Prog}_{sb} \ r \lor \text{is-Ghost}_{sb} \ r \.
from flush-reads-program [OF this] obtain \( O_{h}''''', \mathcal{R}_{h}''''', S_{h}''''' \) \text{acq}_{h}'''''
where (m(a:=v), rws, \( O_{h}'', \mathcal{R}_{h}'', S_{h}'' \)) \rightarrow^{t^*} (m(a:=v), [], \( O_{h}'', \mathcal{R}_{h}'', S_{h}'' \)) by blast
with steps
have (m, sb\_h, \( O_{h}, \mathcal{R}_{h}, S_{h} \)) \rightarrow^{t^*} (m(a:=v), [], \( O_{h}'', \mathcal{R}_{h}'', S_{h}'' \)) by force
with sb Nil show ?thesis by fastforce
qed

todo

lemma bufferd-val-filter-is-Write\_sb-eq-ext:
buffered-val (filter is-Write\_sb \ sb) a = buffered-val sb a
by (induct sb) (auto split: memref.splits)

lemma bufferd-val-filter-is-Write\_sb-eq:
buffered-val (filter is-Write\_sb \ sb) = buffered-val sb
by (rule ext) (rule bufferd-val-filter-is-Write\_sb-eq-ext)

lemma outstanding-refs-is-volatile-Write\_sb-filter-writes:
outstanding-refs is-volatile-Write\_sb (filter is-Write\_sb \ xs) =
outstanding-refs is-volatile-Write\_sb \ xs
by (induct xs) (auto simp add: is-volatile-Write\_sb-def split: memref.splits)

699
A.6 Simulation of Store Buffer Machine without History by Store Buffer Machine with History

**theorem** (in valid-program) concurrent-history-steps-simulates-store-buffer-step:

**assumes** step-sb: \((ts,m,S) \Rightarrow_{sb} (ts',m',S')\)

**assumes** sim: \(i \sim_{h} ts_{h}\)

**shows** \(\exists ts_{h}', S_{h}'. (ts_{h},m,S_{h}) \Rightarrow_{sbh}^* (ts_{h}',m',S_{h}') \land ts' \sim_{h} ts_{h}'\)

**proof**

**interpret** sbh-computation:

computation sbh-memop-step flush-step program-step

\[\lambda p \ p' \text{ is } sb @ [Prog sb p p' is].\]

from step-sb

**show** ?thesis

**proof** (cases rule: concurrent-step-cases)

**case** (Memop i - p is \(\emptyset \) sb \(D' O' R'\) - - \(is' \emptyset' sb' - D' O' R'\))

then obtain

\(ts': ts' = ts[i := (p, is', \emptyset', sb', D', O', R')]\) and

i-bound: \(i < \text{length } ts\) and

\(ts: i = (p, is, \emptyset, sb, D, O, R)\) and

step-sb: \((is, \emptyset, sb, m, D, O, R, S) \rightarrow_{sb}

\((is', \emptyset', sb', m', D', O', R', S')\)

by auto

from sim obtain

lts-eq: \(\text{length } ts = \text{length } ts_{h}\) and

sim-loc: \(\forall i < \text{length } ts. (\exists O' D' R').\)

let \((p, is, \emptyset, sb, D, O, R) = ts_{h}[i]\) in

\(ts[] = (p, is, \emptyset, \text{filter is-Write sb sb}, D', O', R') \land

\((\text{filter is-Write sb sb} = [] \rightarrow sb = [])\))

by cases (auto)

from lts-eq i-bound have i-bound': \(i < \text{length } ts_{h}\)

by simp

from step-sb

**show** ?thesis

**proof** (cases)

**case** (SBReadBuffered a v volatile t)

then obtain

\(is: is = \text{Read volatile a t#is'}\) and

\(O': O'=O\) and

\(S': S'=S\) and

\(R': R'=R\) and

\(D': D'=D\) and

\(m': m'=m\) and

\(\emptyset': \emptyset'=\emptyset(t\rightarrow v)\) and

\(sb': sb' = sb\) and

buf-val: \(\text{buffered-val sb a} = \text{Some v}\)

by auto
from sim-loc [rule-format, OF i-bound] ts-i is
obtain sbh O h Dh where
tsh-i: ts!i = (p, Read volatile a t # is', θ, sbh, Dh, O h, Rh)
and sb: sb = filter is-Write sbh and
sb-empty: filter is-Write sbh = [] → sbh=[]
by (auto simp add: Let-def)
from buf-val
have buf-val': buffered-val sbh a = Some v
by (simp add: bufferd-val-filter-is-Write sbh eq sb)
let ?ts'!i = (p, is', θ(\t\mapsto→ v), sbh, Dh, O h, Rh)
let ?ts' = ts!i := ?ts'!i'
from sbh-memop-step.SBHReadBuffered [OF buf-val']
have (Read volatile a t # is', θ, sbh, m, Dh, O h, S h) \to sbh
(is', θ(t \mapsto→ v), sbh@ [Read sb volatile a t v], m, Dh, O h, S h).
from sbh-computation.Memop [OF i-bound' ts!i this]
have step: (ts!i, m, S h) \Rightarrow sbh (?ts'!i, m, S h).
from sb have sb: sb = filter is-Write sbh (sbh@ [Read sb volatile a t v])
by simp
show ?thesis
proof (cases filter is-Write sbh sb = [])
case False
have ts!i := (p, is', θ(t \mapsto→ v), sbh, Dh, O h, Rh)
by (rule sim-history-config.intros)
using lts-eq
apply simp
using sim-loc i-bound i-bound' sb sb-empty False
apply (auto simp add: nth-list-update)
done
with step show ?thesis
by (auto simp del: fun-upd-apply simp add: S' m' ts' O' θ' D' sb' R')
next
case True
with sb-empty have empty: sbh=[] by simp
from i-bound' have ?ts!i = ?ts!i'
by auto
from sbh-computation.StoreBuffer [OF - this, simplified empty, simplified, OF -
flush-step.Read sb, of m S h] i-bound'
      have (?ts!i', m, S h)
          \Rightarrow sbh (ts!i := (p, is', θ(t \mapsto→ v), [], Dh, O h, Rh)], m, S h)
by (simp add: empty list-update-overwrite)
with step have (ts!i := (p, is', θ(t \mapsto→ v), [], Dh, O h, Rh)], m, S h)

by force
moreover
have ts [i := (p, is', θ(t → v), sb, D, O, R)] \sim_h ts_h[i := (p, is', θ(t → v), [], D_h, O_h, R_h)]
apply (rule sim-history-config.intros)
using lts-eq
apply simp
using lts-eq
apply (auto simp add: Let-def nth-list-update)
done
ultimately show \?thesis
by (auto simp del: fun-upd-apply simp add: S' m' O' D' sb' R' θ'

next
case (SBReadUnbuffered a volatile t)
then obtain
is: is = Read volatile a t#is' and
O': O'=O and
R': R'=R and
S': S'=S and
D': D'=D and
m': m'=m and
\θ': \θ'=\θ(t→m a) and
sb': sb'= sb and
buf: buffered-val sb a = None
by auto
from sim-loc [rule-format, OF i-bound] ts-i is
obtain sb_h O_h R_h D_h where
ts_h-i: ts_h[i = (p, Read volatile a t#is', θ_h, sb_h, D_h, O_h, R_h)] and
sb: sb = filter is-Write sb_h and
sb-empty: filter is-Write sb_h = [] \rightarrow sb_h=[]
by (auto simp add: Let-def)

from buf
have buf': buffered-val sb_h a = None
by (simp add: bufferd-val-filter-is-Write sb-eq sb)

let ?ts_h-i' = (p, is', \θ(t \rightarrow m a), sb_h@[Read_{sb} volatile a t (m a)], D_h, O_h, R_h)
let ?ts_h' = ts_h[i := ?ts_h-i']

from sbh-memop-step.SBHReadUnbuffered [OF buf']
have (Read volatile a t \# is', \θ, sb_h, m, D_h, O_h, R_h, S_h) \rightarrow_{sb_h}
   (is', \θ(t \rightarrow (m a)), sb_h@[Read_{sb} volatile a t (m a)], m, D_h, O_h, R_h, S_h).
from sbh-computation.Memop [OF i-bound' ts_h-i this]
have step: (ts_h, m, S_h) \Rightarrow_{sb_h}
   (?ts_h', m, S_h).
moreover
from sb have sb: sb = filter is-Write_{sb} (sb_h@[Read_{sb} volatile a t (m a)])
by simp
show \( \text{thesis} \)
proof (cases filter is-Write \( sb_h = [] \))
  case False
    have \( ts [i := (p, is', \theta (t \mapsto m a), sb, D, O, R)] \sim_h ts_h' \)
    apply (rule sim-history-config.intros)
    using lts-eq
    apply simp
    using sim-loc i-bound i-bound' sb sb-empty False
    apply (auto simp add: Let-def nth-list-update)
    done
  with step show \( \text{thesis} \)
  by (auto simp del: fun-upd-apply simp add: \( S', m', ts' O' R' D' \theta' sb' \))
next
  case True
  with sb-empty have empty: \( sb_h = [] \) by simp
  from i-bound' have \( ts_h' \| i = ts_h - i' \)
    by auto
  from sbh-computation.StoreBuffer [OF - this, simplified empty, simplified, OF - flush-step.Read sb, of \( m S_h \)] i-bound'
    have \( ts_h [i := (p, is', \theta (t \mapsto m a), [], D_h, O_h, R_h)], m, S_h \)
      by (simp add: empty)
    with step have \( ts_h, m, S_h \rightarrow_{sbh}^* \)
      \( ts_h [i := (p, is', \theta (t \mapsto m a), [], D_h, O_h, R_h)], m, S_h \)
      by force
    moreover
      have \( ts [i := (p, is', \theta (t \mapsto m a), sb, D, O, R)] \sim_h ts_h [i := (p, is', \theta (t \mapsto m a), [], D_h, O_h, R_h)] \)
      apply (rule sim-history-config.intros)
      using lts-eq
      apply simp
      using sim-loc i-bound i-bound' sb empty
      apply (auto simp add: Let-def nth-list-update)
      done
      ultimately show \( \text{thesis} \)
      by (auto simp del: fun-upd-apply simp add: \( S', m', ts' O' R' D' \theta' sb' \))
    qed
next
  case (SBWriteNonVolatile a D f A L R W)
  then obtain
    is: is = Write False a (D, f) A L R W\#is' and
    \( O' \equiv O \) and
    \( R' \equiv R \) and
    \( S' \equiv S \) and
    \( D' \equiv D \) and
    \( m' \equiv m \) and
    \( \theta' \equiv \theta \) and

703
\[ \text{sb}': \text{sb}' = \text{sb}@[\text{Write}_{\text{sb}} \text{False a (D, f)} (f \vartheta) \ A \ L \ R \ W] \]

by auto

\begin{align*}
\text{from sim-loc [rule-format, OF i-bound] ts-i} & \\
\text{obtain sb}_h \text{C}_h \ R_h \ D_h \text{ where} & \\
\text{ts}_h:i: \text{ts}_h:il = (p, \text{Write False a (D,f) A L R W} #is', \vartheta, \text{sb}_h, D_h, C_h, R_h) \text{ and} & \\
\text{sb: sb = filter is-Write}_{\text{sb}} \text{sb}_h & \\
\text{by (auto simp add: Let-def is)} & \\
\end{align*}

\begin{align*}
\text{from sbh-memop-step.SBHWriteNonVolatile} & \\
\text{have (Write False a (D, f) A L R W} #is', \vartheta, \text{sb}_h, D_h, C_h, R_h, \text{S}_h) \rightarrow_{\text{sbh}} & \\
\text{(is', \vartheta, \text{sb}_h @ [Write}_{\text{sb}} \text{False a (D, f)} (f \vartheta) \ A \ L \ R \ W], m, D_h, C_h, R_h, S_h)]. & \\
\text{from sbh-computation.Memop [OF i-bound' ts}_h\text{-i this]} & \\
\text{have (ts}_h, m, \text{S}_h) \Rightarrow_{\text{sbh}} & \\
\text{(ts}_h[i] := (p, is', \vartheta, \text{sb}_h @ [Write}_{\text{sb}} \text{False a (D, f)} (f \vartheta) \ A \ L \ R \ W], D_h, C_h, R_h)], & \\
\text{m, S}_h). & \\
\text{moreover} & \\
\text{have ts [i := (p, is', \vartheta, \text{sb}_h @ [Write}_{\text{sb}} \text{False a (D, f)} (f \vartheta) \ A \ L \ R \ W], D_h, C_h, R_h)] \sim_{\text{h}} & \\
\text{ts}_h[i := (p, is', \vartheta, \text{sb}_h @ [Write}_{\text{sb}} \text{False a (D, f)} (f \vartheta) \ A \ L \ R \ W], D_h, C_h, R_h)]. & \\
\text{apply (rule sim-history-config.intros)} & \\
\text{using lts-eq} & \\
\text{apply simp} & \\
\text{using sim-loc i-bound i-bound' sb} & \\
\text{apply (auto simp add: Let-def nth-list-update)} & \\
\text{done} & \\
\text{ultimately show ?thesis} & \\
\text{by (auto simp add: S'} m' \vartheta' O' R' D' ts' sb') & \\
\text{next} & \\
\text{case (SBWriteVolatile a D f A L R W)} & \\
\text{then obtain} & \\
is: \text{is = Write True a (D, f) A L R W} #is' and & \\
O': O'\subseteq O \text{ and} & \\
R': R'\subseteq R \text{ and} & \\
S': S'\subseteq S \text{ and} & \\
D': D'\subseteq D \text{ and} & \\
m': m'\subseteq m \text{ and} & \\
\vartheta': \vartheta'\subseteq \vartheta \text{ and} & \\
\text{sb': sb'} = \text{sb}@[\text{Write}_{\text{sb}} \text{True a (D, f)} (f \vartheta) \ A \ L \ R \ W] & \\
\text{by auto} & \\
\text{from sim-loc [rule-format, OF i-bound] ts-i is} & \\
\text{obtain sb}_h \text{C}_h \ R_h \ D_h \text{ where} & \\
\text{ts}_h:i: \text{ts}_h:il = (p, \text{Write True a (D,f) A L R W} #is', \vartheta, \text{sb}_h, D_h, C_h, R_h) \text{ and} & \\
\text{sb: sb = filter is-Write}_{\text{sb}} \text{sb}_h & \\
\text{by (auto simp add: Let-def)} & \\
\text{from sbh-computation.Memop [OF i-bound' ts}_h\text{-i SBHWriteVolatile} & \\
\text{have (ts}_h, m, \text{S}_h) \Rightarrow_{\text{sbh}} & \\
\end{align*}
(ts_h[i := (p, is', \emptyset, sb_h) @ Write_{sb} True a (D, f) (f \emptyset) A L R W], True, \mathcal{O}_h, \mathcal{R}_h)], m, \mathcal{S}_h).

moreover
\textbf{have} ts [i := (p, is', \emptyset, sb_h) @ Write_{sb} True a (D, f) (f \emptyset) A L R W]; \mathcal{D}(\mathcal{O}, \mathcal{R})] \sim_h
\textbf{ts}_h[i := (p, is', \emptyset, sb_h) @ Write_{sb} True a (D, f) (f \emptyset) A L R W], True, \mathcal{O}_h, \mathcal{R}_h)]
\textbf{apply} (rule sim-history-config.intros)
\textbf{using} lts-eq
\textbf{apply} simp
\textbf{using} sim-loc i-bound i-bound' sb
\textbf{apply} (auto simp add: Let-def nth-list-update)
done

ultimately \textbf{show} ?thesis
by (auto simp add: ts' _ O' \emptyset' m' sb'D' R' S' )

next
\textbf{case} SBFence
then obtain
is': is = Fence #is' and
O': O'=O and
R': R'=R and
S': S'=S and
D': D'=D and
m': m'=m and
\emptyset': \emptyset'=\emptyset and
sb: sb = [] and
sb': sb' = []
by auto

from sim-loc [rule-format, OF i-bound] ts-i sb is
\textbf{obtain} sb_h \mathcal{O}_h \mathcal{R}_h \mathcal{D}_h where
\textbf{ts}_h-i: \textbf{ts}_h[i := (p, Fence # is', \emptyset, sb_h, \mathcal{D}_h, \mathcal{O}_h, \mathcal{R}_h)] and
sb: [] = filter is-Write_{sb} sb_h
by (auto simp add: Let-def)

from filter-is-Write_{sb}-empty [OF sb [symmetric]]
\textbf{have} \forall r \in \text{set } sb_h. \text{is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r.

from flush-reads-program [OF this] \textbf{obtain} \mathcal{O}_h', \mathcal{S}_h' \mathcal{R}_h' where
\textbf{flsh}: (m, sb_h, \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \rightarrow^{*} (m', [], \mathcal{O}_h', \mathcal{R}_h', \mathcal{S}_h') by blast

let \textbf{?ts}_h' = \textbf{ts}_h[i := (p, Fence # is', \emptyset, [], \mathcal{D}_h, \mathcal{O}_h', \mathcal{R}_h')] from sbh-computation.store-buffer-steps [OF flsh i-bound' ts_h-i]
\textbf{have} (ts_h, m, \mathcal{S}_h) \Rightarrow_{sbh}^{*} (?ts_h', m, \mathcal{S}_h').

also

from i-bound' \textbf{have} i-bound'' : i < \text{length } ?ts_h'
by auto

705
from i-bound' have ts\_h\_i: ts\_h\_i = (p,Fence\#is',\emptyset,[],D_h,O_h',R_h')
by simp
from sbh-computation.Memop [OF i-bound' ts\_h\_i SBHFence] i-bound'
have (?ts\_h\_i, m, S_h') \Rightarrow sbh (ts\_h\_i := (p, is', \emptyset, [], False, O_h',Map.empty)), m,S_h')
by (simp)
finally have (ts\_h\_i, m, S_h') \Rightarrow sbh (ts\_h\_i \{i := p, is', \emptyset, [], False, O_h', Map.empty\}, m, S_h')
moreover
have ts \{i := (p, is', \emptyset, [], False, O_h', Map.empty\}, m, S_h') \Rightarrow sbh
ultimately show \?thesis
by (auto simp add: ts\_h\_O\_h\_R\_h\_D\_h\_S\_h\_R\_w)

next
  case (SBRMWReadOnly cond t a D f ret A L R W)
  then obtain is: is = RMW a t (D, f) cond ret A L R W# is' and
  O\_h': O'\_h'=O and
  R\_h': R'\_h'=R and
  S\_h': S'\_h'=S and
  D\_h': D'\_h'=D and
  m\_h': m'\_h'=m and
  \empty\_\_h': \empty\_\_h'(t \mapsto m a) and
  sb: sb=[] and
  sb\_h': sb\_h'=[] and
  cond: \neg cond (\empty\_\_h'(t \mapsto m a))
by auto
  from sim-loc [rule-format, OF i-bound] ts\_i sb is
  obtain sb\_h (O_h R_h D_h) where
  ts\_h\_i: ts\_h\_i \emptyset = (p,RMW a t (D, f) cond ret A L R W# is',\emptyset, sb\_h, D_h, O_h, R_h) and
  sb: [] = filter is-Write\_sb sb\_h
  by (auto simp add: Let-def)

  from filter-is-Write\_sb-empty [OF sb [symmetric]]
  have \forall r \in set sb\_h, is-Read\_sb r \lor is-Prog\_sb r \lor is-Ghost\_sb r.

  from flush-reads-program [OF this] obtain O_h' S_h' R_h'
  where flsh: (m, sb\_h, O_h, R_h, S_h) \Rightarrow t\_s (m, [], O_h', R_h', S_h') by blast
let $\texttt{ts}_h' = \texttt{ts}_h[i := (p, \text{RMW} a t (D, f) \text{ cond ret } A L R W # \text{is}', \theta, [], D_h, O_h', R_h')]$

from sbh-computation.store-buffer-steps [OF flsh i-bound' ts_h-i]

have $(\texttt{ts}_h, m, S_h) \Rightarrow_{sbh^*} (\texttt{ts}_h', m, S_h')$.

also

from i-bound' have i-bound'': $i < \text{length } \texttt{ts}_h'$

by auto

from i-bound' have $\texttt{ts}_h'\cdot i : \texttt{ts}_h' = (p, \text{RMW} a t (D, f) \text{ cond ret } A L R W # \text{is}', \theta, [], D_h, O_h', R_h')$

by simp

note step= SBHRMWReadOnly [where cond=cond and $\theta=\theta$ and $m=m$, OF cond ]

from sbh-computation.Memop [OF i-bound'' ts_h'-i step ] i-bound'

have $(\texttt{ts}_h', m, S_h') \Rightarrow_{sbh} (\texttt{ts}_h[i := (p, \text{is}', \theta(t \mapsto m a), []), False, O_h', \text{Map.empty}], m, S_h')$

by (simp)

finally

have $(\texttt{ts}_h, m, S_h) \Rightarrow_{sbh^*} (\texttt{ts}_h[i := (p, \text{is}', \theta(t \mapsto m a), []), False, O_h', \text{Map.empty}], m, S_h')$.

moreover

have $\texttt{ts}[i := (p, \text{is}', \theta(t \mapsto m a), [], D, O, R)] \sim_h \texttt{ts}_h[i := (p, \text{is}', \theta(t \mapsto m a), [], False, O_h', \text{Map.empty})], m, S_h']$

apply (rule sim-history-config.intros)

using lts-eq

apply simp

using sim-loc i-bound i-bound' sb

apply (auto simp add: Let-def nth-list-update)

done

ultimately show $\texttt{?thesis}$

by (auto simp add: ts' O' $\theta'$ m' sb' D' S' R')

next

case (SBRMWWrite cond t a D f ret A L R W)

then obtain

is: is = \text{RMW} a t (D, f) \text{ cond ret } A L R W # \text{is}' and

$O': O'=O$ and

$R': R'=R$ and

$S': S'=S$ and

$D': D'=D$ and

m': m'=m(a := f (\theta(t \mapsto (m a)))) and

$\theta': \theta'=\theta(t \mapsto ret (m a) (f (\theta(t \mapsto (m a)))))$ and

sb: sb=[] and

sb': sb' = [] and

cond: cond (\theta(t \mapsto m a))

by auto
from sim-loc [rule-format, OF i-bound] ts-i sb is
obtain sbh O_h R_h D_h acqh where
ts_h-i: ts_h!i = (p,RMW a t (D, f) cond ret A L R W# is',\hat{\theta},sbh,D_h,O_h,R_h) and
sb: [] = filter is-Write sb sb
by (auto simp add: Let-def)

from filter-is-Write sb-empty [OF sb [symmetric]]
have \( \forall r \in \text{set sb} \Rightarrow \text{is-Read sb sb} \lor \text{is-Prog sb sb} \lor \text{is-Ghost sb sb} \).

from flush-reads-program [OF this]
obtain O_h' S_h' R_h'
where flsh: (m, sbh) \rightarrow f*(m, [],O_h',R_h',S_h') by blast

let ?ts_h' = ts_h[i := (p,RMW a t (D, f) cond ret A L R W# is',\hat{\theta},[]),D_h,O_h',R_h']

from sbh-computation.store-buffer-steps [OF flsh i-bound ts_h-i]
have (ts_h, m, S_h) \Rightarrow sbh* (?ts_h', m, S_h').

also

from i-bound' have i-bound'': i < length ?ts_h'
by auto

from i-bound' have ts_h'-i: ?ts_h'i = (p,RMW a t (D, f) cond ret A L R W#is',\hat{\theta},[],D_h,O_h',R_h')
by simp

note step= SBHRMWWrite [where cond=cond and \( \hat{\theta}=\theta \) and m=m, OF cond]
from sbh-computation.Memop [OF i-bound'' ts_h'-i step ] i-bound'

have (?ts_h', m, S_h') \Rightarrow sbh* (ts_h[i := (p, is',
\hat{\theta}(t \mapsto ret (m a)) (f (\hat{\theta}(t \mapsto (m a))))), [], False, O_h' \cup A - R.Map.empty)],
m(a := f (\hat{\theta}(t \mapsto (m a))))),S_h' \oplus W R \in_A L)
by (simp)

finally
have (ts_h, m, S_h) \Rightarrow sbh* (ts_h[i := (p, is',
\hat{\theta}(t \mapsto ret (m a)) (f (\hat{\theta}(t \mapsto (m a))))), [], False, O_h' \cup A - R.Map.empty)],
m(a := f (\hat{\theta}(t \mapsto (m a))))),S_h' \oplus W R \in_A L).

moreover

have ts [i := (p,is',\hat{\theta}(t \mapsto ret (m a)) (f (\hat{\theta}(t \mapsto (m a))))),[],D_h,O_h,R_h] \sim_h
ts_h[i := (p,is',\hat{\theta}(t \mapsto ret (m a)) (f (\hat{\theta}(t \mapsto (m a))))), [],False, O_h' \cup A - R.Map.empty)]
apply (rule sim-history-config.intro)
using lts-eq
apply simp
using sim-loc i-bound i-bound' sb
apply (auto simp add: Let-def nth-list-update)
done

708
ultimately show \( ? \)thesis
by (auto simp add: \( ts' \ O' \ \theta' \ m' \ sb' \ D' \ S' \ \mathcal{R}' \))

next

case (SBGhost A L R W)
then obtain
is: \( is = \text{Ghost A L R W#is'} \) and
\( O': O'\Rightarrow O \) and
\( R': R'\Rightarrow R \) and
\( S': S'\Rightarrow S \) and
\( D': D'\Rightarrow D \) and
\( m': m'\Rightarrow m \) and
\( \theta': \theta'\Rightarrow \theta \) and
\( sb': sb'\Rightarrow sb \)
by auto

from sim-loc [rule-format, OF i-bound] ts-i is
obtain \( sb_h \ O_h \ R_h \ D_h \) where
\( ts_h\i: ts_h\i!i = (p, \text{Ghost A L R W#is'}, \theta, sb_h, D_h, O_h, R_h) \) and
sb: \( sb = \text{filter is-Write}_{sb} \) and
sb-empty: \( \text{filter is-Write}_{sb} \) \( sb_h = [] \)
by (auto simp add: Let-def)

let \( ?ts_h\i' = (p, \text{is'}, \theta, sb_h[@[\text{Ghost}_{sb} A L R W], D_h, O_h, R_h]) \) and
let \( ?ts_h' = ts_h[i := ?ts_h\i'] \)
note step= SBHGhost
from sbh-computation.Memop [OF i-bound' ts_h\i step ] i-bound'
have step: \( (ts_h, m, S_h) \Rightarrow_{sbh} (?ts_h', m, S_h) \)
by (simp)

from sb have sb: \( sb = \text{filter is-Write}_{sb} \) \( sb_h@[\text{Ghost}_{sb} A L R W] \)
by simp

show \( ? \)thesis
proof (cases filter is-Write_{sb} \( sb_h = [] \))

case False

have \( ts[i := (p, \text{is'}, \theta, sb, D, O, R)] \sim_h ?ts_h' \)
apply (rule sim-history-config.intros)
using lts-eq
apply simp
using sim-loc i-bound' sb-empty False
apply (auto simp add: Let-def nth-list-update)
done

with step show \( ? \)thesis
by (auto simp del: fun-upd-apply simp add: \( S' \ m' \ ts' \ O' \ D' \ \theta' \ sb' \mathcal{R}' \))

next

case True
with sb-empty have empty: \( sb_h = [] \) by simp
from i-bound' have \( ?ts_h\i' = ?ts_h\i' \)}
by auto
from sbh-computation.StoreBuffer [OF - this, simplified empty, simplified, OF - flush-step.ghost, of m $S_h$] i-bound'
  have (?ts'h', m, $S_h$)
  \[\Rightarrow_{sbh} (ts'h[i := (p, is', \emptyset, [], D_h, O_h \cup A - R, augment-rels (dom $S_h$) R R_h)], m, $S_h \oplus_W R \oplus_A L)\]
  by (simp add: empty)
with step have (ts'h, m, $S_h$) \[\Rightarrow_{sbh}^* (ts'h[i := (p, is', \emptyset, [], D_h, O_h \cup A - R, augment-rels (dom $S_h$) R R_h)], m, $S_h \oplus_W R \oplus_A L)\]
  by force
moreover
have ts \[i := (p, is', \emptyset, [], D_h, O_h \cup A - R, augment-rels (dom $S_h$) R R_h)\]
  apply (rule sim-history-config.intros)
using lts-eq
apply simp
using sim-loc i-bound i-bound' sb empty
apply (auto simp add: Let-def nth-list-update)
done
ultimately show ?thesis
by (auto simp del: fun-upd-apply simp add: S' m' ts' O' \emptyset' D' sb' R')
qed

next
case (Program i - p is \emptyset D O R p' is')
then obtain
  ts': ts' = ts[i := (p', is@is', \emptyset, sb, D, O, R)] and
  i-bound: i < length ts and
ts-i: ts ! i = (p, is, \emptyset, sb, D, O, R) and
  prog-step: \emptyset \vdash p \rightarrow_{p} (p', is') and
  S': S'\|=S and
  m': m'=m
by auto

from sim obtain
lts-eq: length ts = length ts'h and
sim-loc: \forall i < length ts. (\exists O' D' R').
let (p, is, \emptyset, sb, D, O, R) = ts'h[i in ts!'i=(p, is, \emptyset, filter is-Write sb sb = \emptyset \rightarrow sb = []))
by cases auto

from sim-loc [rule-format, OF i-bound] ts-i
obtain sb'h O_h R_h D_h acq'h where
  ts'h: ts'h[i := (p, is, \emptyset, sb'h, D_h, O_h, R_h)] and
  sb: sb = filter is-Write sb sb' and
  sb-empty: filter is-Write sb sb' = \emptyset \rightarrow sb' = []
by (auto simp add: Let-def)

from lts-eq i-bound have i-bound': i < length ts'h
by simp

let ?ts_h'i' = (p', is @ is', \theta, sb_h @ [Prog sb p p' is'], D_h, O_h, R_h)
let ?ts_h' = ts_h[i := ?ts_h'i']
from sbh-computation.Program [OF i-bound' ts_h'i prog-step]
have step: (ts_h, m, S_h) \Rightarrow sbh (?ts_h', m, S_h).

show ?thesis
proof (cases filter is-Write sb sb_h = [])
  case False
  have ts[i := (p', is@is', \theta, sb,D, O,R)] \sim_h ?ts_h'
    apply (rule sim-history-config.intro)
    using lts-eq
    apply simp
    using sim-loc i-bound' sb False sb-empty
    apply (auto simp add: Let-def nth-list-update)
    done
  with step show ?thesis
  by (auto simp add: ts' S' m')
next
  case True
  with sb-empty have empty: sb_h = [] by simp
  from i-bound' have ?ts_h'i = ?ts_h'i'
    by auto

  from sbh-computation.StoreBuffer [OF - this, simplified empty, simplified, OF - flush-step.Prog sb, of m S_h] i-bound'
  have (?
    ?ts_h', m, S_h)
    \Rightarrow sbh (ts_h[i := (p', is@is', \theta, sb,D, O,R)], m, S_h)
    by (simp add: empty)
  with step have (ts_h, m, S_h) \Rightarrow sbh
    (ts_h[i := (p', is@is', \theta, [], D_h, O_h, R_h)], m, S_h)
    by force
  moreover
  have ts[i := (p', is@is', \theta, sb,D, O,R)] \sim_h ts_h[i := (p', is@is', \theta, [], D_h, O_h, R_h)]
    apply (rule sim-history-config.intro)
    using lts-eq
    apply simp
    using sim-loc i-bound' sb empty
    apply (auto simp add: Let-def nth-list-update)
    done
  ultimately show ?thesis
  by (auto simp del: fun-upd-apply simp add: S' m' ts')
qed
next
  case (StoreBuffer i - p is \emptyset sb D O R - - - sb'O R')
  then obtain ts': ts'[i := (p, is,\emptyset, sb', D, O', R')] and
  i-bound: i < length ts and
ts-i: ts ! i = (p, is, ∅, sb, D, O, R) and
sb-step: (m, sb, O, R, S) → (m', sb', O', R', S') by auto

from sim obtain
lts-eq: length ts = length ts and
sim-loc: ∀ i < length ts. (∃ O', D', R'. ts ! i = ts ![i] = (p, is, ∅, filter is-Write sb sb, D', O', R')) ∧
(filter is-Write sb sb = [] −→ sb = [] ) by auto from sim-loc

obtain sb-h O-h R-h D-h acq-h where

from sim-loc [rule-format, OF i-bound] ts-i

obtain sb-h O-h R-h D-h where

from lts-eq i-bound have i-bound': i < length ts

by simp

from flush-simulates-filter-writes [OF sb-step sb, OF O-h R-h S-h]

obtain sb-h' O-h R-h' S-h' where

from flush-i-bound sb-step obtain volatile a sop v A L R W where sb=Write sb volatile a sop v A L R W#sb'

by cases force

moreover

have ts ![i] := (p, is, ∅, sb', D, O', R') −→ ts ![i] := (p, is, ∅, sb'h, D', O'h, R'h')

apply (rule sim-history-config.intros)
using lts-eq
apply simp
using sim-loc i-bound i-bound' sb sb' sb'-empty
apply (auto simp add: Let-def nth-list-update)
done

ultimately show ?thesis

by (auto simp add: ts')

qed

qed
theorem (in valid-program) concurrent-history-steps-simulates-store-buffer-steps:
assumes step-sb: \((ts, m, S') \Rightarrow_{sb}^* (ts', m', S')\)
shows \(\exists tsh S_h. ts \sim_h tsh \implies \exists tsh' S_{h'}. (tsh, m, S_h) \Rightarrow_{sbh}^* (tsh', m', S_{h'}) \land ts' \sim_h tsh'\)
using step-sb
proof (induct rule: converse-rtranclp-induct-sbh-steps)
case refl thus ?case by auto
next
case (step ts m S ts'' m'' S'')
have first: \((ts, m, S) \Rightarrow_{sb}^* (ts'', m'', S'')\) by fact
have sim: ts \sim_h tsh by fact
from concurrent-history-steps-simulates-store-buffer-step [OF first sim, of \(S_h\)]
obtain \(tsh'' S_{h''}\) where
  exec: \((tsh, m, S_h) \Rightarrow_{sbh}^* (tsh'', m'', S_{h''})\) and sim: ts'' \sim_h tsh''
  by auto
from step.hyps (3) [OF sim, of \(S_h''\)]
obtain \(tsh_h' S_{h'}\) where exec-rest: \((tsh_h'' m'' S_{h''}) \Rightarrow_{sbh}^* (tsh_h', m', S_{h'})\) and sim': ts' \sim_h tsh_h'
  by auto
note exec also note exec-rest
finally show ?case
using sim' by blast
qed

theorem (in xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-execution:
assumes exec-sb: \((ts_{sb}, m_{sb}, x) \Rightarrow_{sb}^* (ts_{sb}', m_{sb}', x')\)
assumes init: initial \(ts_{sb} S_{sb}\)
assumes valid: valid ts_{sb}
assumes sim: \((ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)\)
assumes safe: safe-reach-direct safe-free-flowing \((ts, m, S)\)
shows \(\exists tsh_h' tsh' m' S'.\)
\((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbh}^* (tsh_h', m_{sb}', S_{h'}) \land tsh_{sb} \sim_h tsh_h' \land (ts, m, S) \Rightarrow_{d}^* (ts', m', S') \land (tsh_h', m_{sb}', S_{h'}) \sim (ts', m', S')\)
proof –
from init interpret ini: initial \(ts_{sb} S_{sb}\).
from concurrent-history-steps-simulates-store-buffer-steps [OF exec-sb ini.history-refl, of \(S_{sb}\)]
obtain \(tsh_h' S_{h'}\) where
  sbh: \((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbh}^* (tsh_h', m_{sb}', S_{h'})\) and
  sim-sbh: \(tsh_{sb} \sim_h tsh_h'\)
  by auto
from concurrent-direct-execution-simulates-store-buffer-history-execution [OF sbh init valid sim safe]
obtain \(tsh' m' S'\) where
  d: \((ts, m, S) \Rightarrow_{d}^* (ts', m', S')\) and
  d-sim: \((tsh_h', m_{sb}', S_{h'}) \sim (ts', m', S')\)
  by auto
with sbh sim-sbh show ?thesis by blast

713
**inductive** sim-direct-config::

\[
(p, p '\text{store-buffer}', '\text{dirty}', '\text{owns}', '\text{rels}) \rightarrow \text{thread-config list} \Rightarrow (p, '\text{unit}', '\text{rels}')
\]

\[
\text{thread-config list} \Rightarrow \text{bool}
\]

\[
(- \sim [60,60] 100)
\]

**where**

\[
\text{length ts} = \text{length ts}_d;
\]

\[
\forall i < \text{length ts}.
\]

\[
(\exists O' D' R').
\]

\[
\text{let } (p, \text{is}, 0, \text{sb}, D, O, R) = ts_d[i] \text{ in}
\]

\[
ts_d[i] = (p, \text{is}, 0, [], D', O', R')
\]

\[
\Rightarrow
\]

\[
ts \sim ts_d
\]

**lemma** empty-sb-sims:

**assumes** empty:

\[
\forall i. p \text{ is xs sb } D O R. i < \text{length ts}_s \rightarrow ts_s[i] = (p, \text{is}, \text{xs}, \text{sb}, D, O, R) \rightarrow \text{sb} = []
\]

**assumes** sim-h: ts_s \sim h ts_h

**assumes** sim-d: (ts_h, m_h, S_h) \sim (ts, m, S)

**shows** ts_s \sim ts \land m_h = m \land \text{length ts}_s = \text{length ts}

**proof**

\[
\text{from sim-h empty}
\]

**have** empty':

\[
\forall i. p \text{ is xs sb } D O R. i < \text{length ts}_h \rightarrow ts_h[i] = (p, \text{is}, \text{xs}, \text{sb}, D, O, R) \rightarrow \text{sb} = []
\]

**apply** (cases)

**apply** clarsimp

**subgoal for** i

**apply** (drule-tac x=i in spec)

**apply** (auto simp add: Let-def)

**done**

**done**

**from** sim-h sim-config-emptyE [OF empty' sim-d]

**show** ?thesis

**apply** cases

**apply** clarsimp

**apply** (rule sim-direct-config.intros)

**apply** clarsimp

**apply** clarsimp

**using** empty'

**subgoal for** i

**apply** (drule-tac x=i in spec)

**apply** (drule-tac x=i in spec)

**apply** (auto simp add: Let-def)

**done**

**done**

714
qed

lemma empty-d-sims:
assumes sim: ts_{sb} \sim_d ts
shows \exists ts_h. ts_{sb} \sim_h ts_h \land (ts_h,m,S) \sim (ts,m,S)
proof
from sim obtain
leq: length ts_{sb} = length ts and
sim: \forall i < length ts_{sb}.
(\exists O'D'R'.
let (p,is, @,sb,D,O,R) = ts_{sb}\langle i\rangle in
 ts_{sb}\langle i\rangle=(p,is, @, [], D',O',R'))
by cases auto
define ts_h where
 ts_h \equiv \map {\lambda (p,is, @,sb,D,O,R). (p,is, @, [],' a memref list,D,O,R)}
ts have ts_{sb} \sim_h ts_h
apply (rule sim-history-config.intros)
using leq sim
apply (auto simp add: ts_h-def Let-def leq)
done
moreover have empty:
\forall i p is xs sb D O R. i < length ts_h \rightarrow ts_h\langle i\rangle=(p,is,xs,sb,D,O,R) \rightarrow sb=[]
apply (clarsimp simp add: ts_h-def Let-def)
subgoal for i
apply (case_tac ts_{sb}\langle i\rangle)
apply auto
done
done
have (ts_h,m,S) \sim (ts,m,S)
apply (rule sim-config-emptyI [OF empty])
apply (clarsimp simp add: ts_h-def Let-def)
subgoal for i
apply (case_tac ts_{sb}\langle i\rangle)
apply auto
done
done
ultimately show ?thesis by blast
qed

theorem (in xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-execution-empty:
assumes exec-sb: (ts_{sb},m_{sb},x) \Rightarrow^{*}_{sb} (ts_{sb}',m_{sb}',x')
assumes init: initial_{sb} ts_{sb} S_{sb}
assumes valid: valid ts_{sb}
assumes empty:
\forall i p is xs sb D O R. i < length ts_{sb}' \rightarrow ts_{sb}'\langle i\rangle=(p,is,xs,sb,D,O,R) \rightarrow sb=[]
assumes sim: (ts_{sb},m_{sb},S_{sb}) \sim (ts,m,S)
assumes safe: safe-reach-direct safe-free-flowing (ts,m,S)
shows \( \exists ts' S'. \)
\[ (ts,m,S) \Rightarrow_d^* (ts',m_{sb}',S') \land ts_{sb}' \sim_d ts' \]
proof –
from concurrent-direct-execution-simulates-store-buffer-execution [OF exec-sb init valid sim safe]

obtain ts_{sh}', S_{sh}' ts' m' S' where
\[ (ts_{sb},m_{sb},S_{sb}) \Rightarrow_{sh}^* (ts_{sh}',m_{sb}',S_{sh}') \land \]
sim-h: ts_{sb}' \sim_h ts_{sh}' and
exec: (ts,m,S) \Rightarrow_d^* (ts',m',S') and
sim: (ts_{sh}',m_{sb}',S_{sh}') \sim (ts',m',S')
by auto
from empty-sb-sims [OF empty sim-h sim]

obtain ts_{sb}' \sim_d ts' m_{sb}' = m' length ts_{sb}' = length ts'
by auto
thus \(?thesis \)
using exec
by blast
qed

locale initial_d = simple-ownership-distinct + read-only-unowned + unowned-shared +
fixes valid
assumes empty-is: \([i < \text{length ts}; ts!i=(p,is,xs,sb,D,O,R)] \Rightarrow is=[]\]
assumes empty-rels: \([i < \text{length ts}; ts!i=(p,is,xs,sb,D,O,R)] \Rightarrow R=\text{Map.empty}\]
assumes valid-init: valid (map (\(\lambda (p,is,\emptyset ,sb,D,O,R)) \text{ ts})

locale empty-store-buffers =
fixes ts::(\(p,p\text{ store-buffer},\text{bool.owns},\text{rels}) \text{ thread-config list}
assumes empty-sb: \([i < \text{length ts}; ts!i=(p,is,xs,sb,D,O,R)] \Rightarrow sb=[]\]

lemma initial-d-sb:
assumes init: initial_d ts S valid
shows init\(_{sb}\) (map (\(\lambda(p,is,\emptyset ,sb,D,O,R)) \text{ ts}) S)
(is init\(_{sb}\) ?map S)
proof –
from init interpret ini: initial_d ts S .
show \(?thesis \)
proof (intro-locales)
show simple-ownership-distinct \(?map\)
apply (clarsimp simp add: simple-ownership-distinct-def)
subgoal for i j
apply (case-tac ts!i)
apply (case-tac ts!j)
apply (cut-tac i=i and j=j in ini.simple-ownership-distinct)
apply clarsimp
apply clarsimp
apply clarsimp
apply clarsimp
apply clarsimp
apply assumption
apply assumption
apply auto

716
done
done

next
show read-only-unowned $S$ ?map
apply (clarsimp simp add: read-only-unowned-def)
subgoal for i
apply (case-tac ts!i)
apply (cut-tac i=i in ini.read-only-unowned)
apply clarsimp
apply assumption
apply auto
done
done

next
show unowned-shared $S$ ?map
apply (clarsimp simp add: unowned-shared-def')
apply (rule ini.unowned-shared')
apply clarsimp
subgoal for a i
apply (case-tac ts!i)
apply auto
done
done

next
show initial$sb$-axioms ?map
apply (unfold-locales)
subgoal for i
apply (case-tac ts!i)
apply simp
done
subgoal for i
apply (case-tac ts!i)
apply clarsimp
apply (rule-tac i=i in ini.empty-is)
apply clarsimp
apply fastforce
done
subgoal for i
apply (case-tac ts!i)
apply clarsimp
apply (rule-tac i=i in ini.empty-rels)
apply clarsimp
apply fastforce
done
done
done
qed
qed

theorem (in xvalid-program-progress) store-buffer-execution-result-sequential-consistent:
assumes exec-$sb$: $(ts_{sb}, m, x) \Rightarrow_{sb}^{*} (ts_{sb}', m', x')$
assumes empty': empty-store-buffers ts_{sb}'
assumes sim: ts_{sb} \sim_d ts
assumes init: initial_d ts S valid
assumes safe: safe-reach-direct safe-free-flowing (ts,m,S)
shows \exists ts' S'.
    (ts,m,S) \Rightarrow_d (ts',m',S') \land ts_{sb}' \sim_d ts'
proof –
from empty'
have empty':
\forall i \ p is xs sb D O R. i < length ts_{sb}' \rightarrow ts_{sb}'!i=(p,is,xs,sb,D,O,R)\rightarrow sb=[]
    by (auto simp add: empty-store-buffers-def)
define ts_h where ts_h \equiv map (\lambda (p,is,\theta,sb,D,O,R).
\forall i \ p is xs sb D' O' R'. i < length ts_{sb}' \rightarrow ts_{sb}'!i=(p,is,xs,sb,D',O',R'))
ts
from initial-d-sb [OF init]
have init-h: initial_{sb} ts_h S
    by (simp add: ts_h-def)
from initial_d.valid-init [OF init]
have valid-h: valid ts_h
    by (simp add: ts_h-def)
from sim obtain
leq: length ts_{sb} = length ts and
sim: \forall i < length ts_{sb}.
    (\exists O' D' R').
        let (p,is,\theta,sb,D,O,R) = ts!i in
        ts_{sb}'!i=(p,is,\theta,[],D',O',R'))
    by cases auto
have sim-h: ts_{sb} \sim_h ts_h
    apply (rule sim-history-config.intros)
    using leq sim
    apply (auto simp add: ts_h-def Let-def leq)
done

from concurrent-history-steps-simulates-store-buffer-steps [OF exec-sb sim-h, of S]
obtain ts_h' S_h' where steps-h: (ts_h,m,S) \Rightarrow_{sbh} (ts_h',m',S_h') and
    sim-h': ts_{sb}' \sim_h ts_h'
    by auto
moreover
have empty:
\forall i \ p is xs sb D O R. i < length ts_h \rightarrow ts_h!i=(p,is,xs,sb,D,O,R)\rightarrow sb=[]
    apply (clarsimp simp add: ts_h-def Let-def)
subgoal for i
    apply (case-tac ts!i)
    apply auto
done
done

have sim': (ts_h,m,S) \sim (ts,m,S)
    apply (rule sim-config-emptyI [OF empty])

718
apply (clarsimp simp add: ts\_h-def)
apply (clarsimp simp add: ts\_h-def Let-def)
subgoal for i
apply (case-tac ts\_i)
apply auto
done
done

done

from concurrent\-direct\-execution\-simulates\-store\-buffer\-history\-execution [OF steps-h init-h valid-h sim\_'safe]
obtain ts\_m'' S'' where steps: (ts, m, S) \Rightarrow_d^* (ts\_'m'', S'')
and sim': (ts\_h', m', S\_h') \sim (ts\_'m'', S'')
by blast
from empty\-sb\-sims [OF empty\_'sim-h sim\_'h'] steps
show ?thesis
by auto
qed

locale initial_v = simple\-ownership\-distinct + read\-only\-unowned + unowned\-shared + fixes valid
assumes empty\-is: \[i < \text{length ts}; ts!i=(p, is, xs, sb, D, O, R)\] \Rightarrow is=\[
assumes valid\-init: valid (map (λ (p, is, sb, D, O, R)). (p, is, [], D, O, Map.empty)) ts)

theorem (in xvalid\-program\-progress) store\-buffer\-execution\-result\-sequential\-consistent':
assumes exec\-sb: (ts\_sb\_m, x) \Rightarrow_{sb}^* (ts\_sb\_'m', x')
assumes empty': empty\-store\-buffers ts\_sb'
assumes sim: ts\_sb \sim_d ts
assumes init: initial_v ts S valid
assumes safe: safe\-reach\-virtual safe\-free\-flowing (ts, m, S)
shows \exists ts' S'.
\quad (ts, m, S) \Rightarrow_{v'}^* (ts\_'m', S') \land ts\_sb' \sim_d ts'
proof -
define ts\_d where ts\_d == (map (λ (p, is, sb, D, O, R')). (p, is, [], D, O, Map.empty::rels)) ts)
have rem-ts: remove\-rels ts\_d = ts
  apply (rule nth\-equalityI)
apply (simp add: ts\_d-def remove-rels-def)
apply (clarsimp simp add: ts\_d-def remove-rels-def)
subgoal for i
apply (case-tac ts\_i)
apply clarsimp
done
done
from sim
have sim': ts\_sb \sim_d ts_d

719
apply cases
apply (rule sim-direct-config.intros)
apply (auto simp add: tsd_def)
done

have init': initial\_d ts\_d S valid
proof (intro-locales)
  from init have simple-ownership-distinct ts
    by (simp add: initial\_v-def)
  then
  show simple-ownership-distinct ts\_d
    apply (clarsimp simp add: tsd_def simple-ownership-distinct-def)
    subgoal for i j
    apply (case-tac ts\_i)
    apply (case-tac ts\_j)
    apply force
    done
  done

next
  from init have read-only-unowned S ts
    by (simp add: initial\_v-def)
  then show read-only-unowned S ts\_d
    apply (clarsimp simp add: tsd_def read-only-unowned-def)
    subgoal for i
    apply (case-tac ts\_i)
    apply force
    done
  done

next
  from init have unowned-shared S ts
    by (simp add: initial\_v-def)
  then
  show unowned-shared S ts\_d
    apply (clarsimp simp add: tsd_def unowned-shared-def)
    apply force
    done
  done

next
  have eq: ((\(\lambda(p, is, \emptyset, sb, D, O, R)\). (p, is, \emptyset, [], D, O, R)) \circ
    (\(\lambda(p, is, \emptyset, sb, D, O, R')\). (p, is, \emptyset, (), D, O, Map.empty))) =
    (\(\lambda(p, is, \emptyset, sb, D, O, R')\). (p, is, \emptyset, [], D, O, Map.empty))
  apply (rule ext)
  subgoal for x
  apply (case-tac x)
  apply auto
  done
  done
  from init have initial\_v-axioms ts valid
    by (simp add: initial\_v-def)
  then
show initial_d-axioms ts_d valid
apply (clarsimp simp add: ts_d-def initial_v-axioms-def initial_d-axioms-def eq)
apply (rule conjI)
apply clarsimp
subgoal for i
apply (case-tac ts_d!i)
apply force
done
apply clarsimp
subgoal for i
apply (case-tac ts_d!i)
apply force
done
done
qed
{
fix ts_d' m' S'
assume exec: (ts_d', m, S) ⇒ d⋆ (ts_d', m', S')
have safe-free-flowing (ts_d', m', S')
proof -
from virtual-simulates-direct-steps [OF exec]
have exec-v: (ts, m, S) ⇒ v⋆ (remove-rels ts_d', m', S')
by (simp add: rem-ts)
have eq: map (owned ◦ (λ (p, is, θ, sb, D, O, R). (p, is, θ, (), D, O, ())))
  ts_d' = map owned ts_d'
by auto
from exec-v safe
have safe-free-flowing (remove-rels ts_d', m', S')
by (auto simp add: safe-reach-def)
thен show ?thesis
by (auto simp add: safe-free-flowing-def remove-rels-def owned-def eq)
qed
}
hence safe': safe-reach-direct safe-free-flowing (ts_d, m, S)
by (simp add: safe-reach-def)

from store-buffer-execution-result-sequential-consistent [OF exec-sb empty' sim' init' safe']
obtain ts_d' S' where
exec-d: (ts_d, m, S) ⇒ d⋆ (ts_d', m', S') and sim-d: ts_{sb}' ~_d ts_d'
by blast

from virtual-simulates-direct-steps [OF exec-d]
have (ts, m, S) ⇒ v⋆ (remove-rels ts_d', m', S')
by (simp add: rem-ts)
moreover
from sim-d
have ts_{sb}' ~_d remove-rels ts_d'
apply (cases)
apply (rule sim-direct-config.intros)
apply (auto simp add: remove-rels-def)
done
ultimately show ?thesis
  by auto
qed

A.7 Plug Together the Two Simulations

corollary (in valid-program) concurrent-direct-steps-simulates-store-buffer-step:
assumes step-sb: \((ts_{sb}, m_{sb}, \mathcal{S}_{sb}) \Rightarrow_{sb} (ts_{sb}', m_{sb}', \mathcal{S}_{sb}')\)
assumes sim-h: \(ts_{sb} \sim_h ts_{sbb}\)
assumes valid-own: valid-ownership \(\mathcal{S}_{sbh} ts_{sb}\)
assumes valid-sb-reads: valid-reads \(m_{sb} ts_{sbh}\)
assumes valid-hist: valid-history program-step \(ts_{sbh}\)
assumes valid-sharing: valid-sharing \(\mathcal{S}_{sbh} ts_{sbh}\)
assumes tmps-distinct: tmps-distinct \(ts_{sbh}\)
assumes valid-sops: valid-sops \(ts_{sbh}\)
assumes valid-dd: valid-data-dependency \(ts_{sbh}\)
assumes load-tmps-fresh: load-tmps-fresh \(ts_{sbh}\)
assumes enough-flushs: enough-flushs \(ts_{sbh}\)
assumes valid-program-history: valid-program-history \(ts_{sbh}\)
assumes valid: valid \(ts_{sbh}\)
assumes safe-reach: safe-reach-direct safe-delayed \((ts, m, \mathcal{S})\)
shows \(\exists ts_{sbh}', \mathcal{S}_{sbh}'\).
\((ts_{sbh}, m_{sb}, \mathcal{S}_{sbh}) \Rightarrow_{sbb} (ts_{sbh}', m_{sb}', \mathcal{S}_{sbh}') \land ts_{sb}' \sim_h ts_{sbh}' \land \)
valid-ownership \(\mathcal{S}_{sbh}' ts_{sbh}' \land valid-reads m_{sb}' ts_{sbh}' \land \)
valid-history program-step \(ts_{sbh}' \land \)
valid-sharing \(\mathcal{S}_{sbh'} ts_{sbh}' \land tmpps-distinct ts_{sbh}' \land valid-data-dependency ts_{sbh}' \land \)
valid-sops \(ts_{sbh}' \land load-tmps-fresh ts_{sbh}' \land enough-flushs ts_{sbh}' \land \)
valid-program-history \(ts_{sbh}' \land valid ts_{sbh}' \land \)
\((\exists ts', \mathcal{S}', m'. \Rightarrow_d (ts', m', \mathcal{S}')) \land \)
\((ts_{sbh}', m_{sb}', \mathcal{S}_{sbh}') \sim (ts', m', \mathcal{S}'))\)
proof –
from concurrent-history-steps-simulates-store-buffer-step [OF step-sb sim-h]
  obtain \(ts_{sbh}', \mathcal{S}_{sbh}'\) where
  steps-h: \((ts_{sbh}, m_{sb}, \mathcal{S}_{sbh}) \Rightarrow_{sbh} (ts_{sbh}', m_{sb}', \mathcal{S}_{sbh}') \land ts_{sb}' \sim_h ts_{sbh}'\)
  sim-h': \(ts_{sb} \sim_h ts_{sbh}'\)
  by blast
moreover
from concurrent-direct-steps-simulates-store-buffer-history-steps [OF steps-h valid-own valid-sb-reads valid-hist valid-sharing tmpps-distinct valid-sops valid-dd load-tmps-fresh enough-flushs valid-program-history valid sim safe-reach]
  obtain \(m' ts' \mathcal{S}'\) where
  \((ts, m, \mathcal{S}) \Rightarrow_d (ts', m', \mathcal{S}') \land (ts_{sbh}', m_{sb}', \mathcal{S}_{sbh}') \sim (ts', m', \mathcal{S}'))\)
valid-ownership \(\mathcal{S}_{sbh'} ts_{sbh'} \land valid-reads m_{sb'} ts_{sbh'} \land valid-history program-step ts_{sbh'}\)
valid-sharing \(\mathcal{S}_{sbh'} ts_{sbh'} \land tmpps-distinct ts_{sbh'} \land valid-data-dependency ts_{sbh'}\)
valid-sops \(ts_{sbh'} \land load-tmps-fresh ts_{sbh'} \land enough-flushs ts_{sbh'}\)
valid-program-history ts_{sbh}' \rightarrow \text{valid ts}_{sbh}'

by blast
ultimately
show \text{thesis}
by blast
qed

lemma conj-commI: P \land Q \rightarrow Q \land P
by simp
lemma def-to-eq: P = Q \rightarrow P \equiv Q
by simp

custom \text{xvalid-program}
begin

definition
invariant ts S m ≡
valid-ownership S ts \land valid-reads m ts \land valid-history program-step ts \land
valid-sharing S ts \land tmps-distinct ts \land valid-data-dependency ts \land
valid-sops ts \land load-tmps-fresh ts \land enough-flushs ts \land valid-program-history ts \land
valid ts

definition ownership-inv ≡ valid-ownership
definition sharing-inv ≡ valid-sharing
definition temporaries-inv ts ≡ tmps-distinct ts \land load-tmps-fresh ts

definition history-inv ts m ≡ valid-history program-step ts \land valid-program-history ts \land
valid-reads m ts
definition data-dependency-inv ts ≡ valid-data-dependency ts \land load-tmps-fresh ts \land
valid-sops ts

definition barrier-inv ≡ enough-flushs

lemma invariant-grouped-def: invariant ts S m ≡
ownership-inv S ts \land sharing-inv S ts \land temporaries-inv ts \land data-dependency-inv ts \land
history-inv ts m \land barrier-inv ts \land valid ts

apply (rule def-to-eq)
apply (auto simp add: ownership-inv-def sharing-inv-def barrier-inv-def temporaries-inv-def history-inv-def data-dependency-inv-def invariant-def)
done

theorem (in \text{xvalid-program}) simulation':
assumes step-sb: (ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbh} (ts_{sb}', m_{sb}', S_{sb}')
assumes sim: (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)
assumes inv: invariant ts_{sb} S_{sb} m_{sb}
assumes safe-reach: safe-reach-direct safe-delayed (ts, m, S)
shows invariant ts_{sb}' S_{sb}' m_{sb}' \land
(\exists ts' S' m'. (ts, m, S) \Rightarrow_{d^*} (ts', m', S') \land (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S'))

using inv sim safe-reach

723
apply (unfold invariant-def)
apply (simp only: conj-assoc)
apply (rule concurrent-direct-steps-simulates-store-buffer-history-step [OF step-sb])
apply simp-all
done

lemmas (in xvalid-program) simulation = conj-commI [OF simulation]
end
end

A.8 PIMP

theory PIMP
imports ReduceStoreBufferSimulation
begin

datatype expr = Const val | Mem bool addr | Tmp sop
  | Unop val ⇒ val expr
  | Binop val ⇒ val ⇒ val expr expr

datatype stmt =
  Skip
  | Assign bool expr expr exprs ⇒ owns exprs ⇒ owns exprs ⇒ owns exprs ⇒ owns
  | CAS expr expr exprs ⇒ owns exprs ⇒ owns exprs ⇒ owns exprs ⇒ owns
  | Seq stmt stmt
  | Cond expr stmt stmt
  | While expr stmt
  | SGhost exprs ⇒ owns exprs ⇒ owns exprs ⇒ owns exprs ⇒ owns
  | SFence

primrec used-tmps:: expr ⇒ nat — number of temporaries used
where
used-tmps (Const v) = 0
| used-tmps (Mem volatile addr) = 1
| used-tmps (Tmp sop) = 0
| used-tmps (Unop f e) = used-tmps e
| used-tmps (Binop f e1 e2) = used-tmps e1 + used-tmps e2

primrec issue-expr:: tmp ⇒ expr ⇒ instr list — load operations
where
issue-expr t (Const v) = []
| issue-expr t (Mem volatile a) = [Read volatile a t]
| issue-expr t (Tmp sop) = []
| issue-expr t (Unop f e) = issue-expr t e
| issue-expr t (Binop f e1 e2) = issue-expr t e1 @ issue-expr (t + (used-tmps e1)) e2
primrec eval-expr:: tmp ⇒ expr ⇒ sop — calculate result
  where
  eval-expr t (Const v) = ((t),λθ. v)
  eval-expr t (Mem volatile a) = ((t),λθ. the (θ t))
  eval-expr t (Tmp sop) = sop
  — trick to enforce sop to be sensible in the current context, without
  having to include wellformedness constraints
  eval-expr t (Unop f e) = (let (D, f e) = eval-expr t e in (D,λθ. f (f e θ))))
  eval-expr t (Binop f e_1 e_2) = (let (D_1, f_1) = eval-expr t e_1;
                               (D_2, f_2) = eval-expr (t + (used-tmps e_1)) e_2
                               in (D_1 ∪ D_2,λθ. f (f_1 θ) (f_2 θ))))

primrec valid-sops-expr:: nat ⇒ expr ⇒ bool
  where
  valid-sops-expr t (Const v) = True
  valid-sops-expr t (Mem volatile a) = True
  valid-sops-expr t (Tmp sop) = ((∀t′ ∈ fst sop. t′ < t) ∧ valid-sop sop)
  valid-sops-expr t (Unop f e) = valid-sops-expr t e
  valid-sops-expr t (Binop f e_1 e_2) = (valid-sops-expr t e_1 ∧ valid-sops-expr t e_2)

primrec valid-sops-stmt:: nat ⇒ stmt ⇒ bool
  where
  valid-sops-stmt t Skip = True
  valid-sops-stmt t (Assign volatile a e A L R W) = (valid-sops-expr t a ∧ valid-sops-expr t e)
  valid-sops-stmt t (CAS a c e s e A L R W) = (valid-sops-expr t a ∧ valid-sops-expr t c ∧
                                          valid-sops-expr t s)
  valid-sops-stmt t (Seq s_1 s_2) = (valid-sops-stmt t s_1 ∧ valid-sops-stmt t s_2)
  valid-sops-stmt t (Cond e s_1 s_2) = (valid-sops-expr t e ∧ valid-sops-stmt t s_1 ∧
                                      valid-sops-stmt t s_2)
  valid-sops-stmt t (While e s) = (valid-sops-expr t e ∧ valid-sops-stmt t s)
  valid-sops-stmt t (SGhost A L R W) = True
  valid-sops-stmt t SFence = True

type-synonym stmt-config = stmt × nat
consts isTrue:: val ⇒ bool

inductive stmt-step:: tmps ⇒ stmt-config ⇒ stmt-config × instrs ⇒ bool
  (- ⊢ - →_s - [60,60,60] 100)
  for θ
  where
    AssignAddr:
    ∀sop. a ≠ Tmp sop ⇒
    θ ⊢ (Assign volatile a e A L R W, t) →_s
\[(\text{Assign volatile (Tmp (eval-expr t a)) e A L R W, } t + \text{used-tmps a), issue-expr t a})\]

<table>
<thead>
<tr>
<th>Assign:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D \subseteq \text{dom } \varnothing \implies \varnothing \vdash (\text{Assign volatile (Tmp (D,a)) e A L R W, } t) \rightarrow_s)</td>
</tr>
<tr>
<td>(((\text{Skip, } t + \text{used-tmps } e), \text{issue-expr } t e \varnothing[\text{Write volatile (a } \varnothing) (\text{eval-expr } t e) (A \varnothing) (L \varnothing) (R \varnothing) (W \varnothing)]))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CASAddr:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\forall \text{sop. } a \neq \text{Tmp sop } \implies \varnothing \vdash (\text{CAS a c e s A L R W, } t) \rightarrow_s)</td>
</tr>
<tr>
<td>(((\text{CAS (Tmp (eval-expr t a)) c e s A L R W, } t + \text{used-tmps a), issue-expr t a}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CASComp:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\forall \text{sop. } c \neq \text{Tmp sop } \implies \varnothing \vdash (\text{CAS (Tmp (D_a,a)) c e s A L R W, } t) \rightarrow_s)</td>
</tr>
<tr>
<td>(((\text{CAS (Tmp (D_a,a)) (Tmp (eval-expr t c e)) s e A L R W, } t + \text{used-tmps c e}), \text{issue-expr t c e}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CAS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>([D_a \subseteq \text{dom } \varnothing; D_c \subseteq \text{dom } \varnothing; \text{eval-expr } t s_e = (D,f)] \implies \varnothing \vdash (\text{CAS (Tmp (D_a,a)) (Tmp (D_c,c)) s e A L R W, } t) \rightarrow_s)</td>
</tr>
<tr>
<td>(((\text{Skip, Suc (t + } \text{used-tmps } s_e)), \text{issue-expr } t s_e \varnothing[\text{RMW (a } \varnothing) (t + \text{used-tmps } s_e) (D,f) (\lambda } \varnothing. \text{the (\varnothing (t + } \text{used-tmps } s_e)) = c \varnothing) (\lambda v_1 v_2, v_1) [A \varnothing) (L \varnothing) (R \varnothing) (W \varnothing)]))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seq:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varnothing \vdash (s_1, t) \rightarrow_s ((s'_1, t'), \text{is}) \implies \varnothing \vdash (\text{Seq } s_1 s_2, t) \rightarrow_s ((\text{Seq } s'_1 s_2, t'),\text{is}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SeqSkip:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varnothing \vdash (\text{Seq Skip } s_2, t) \rightarrow_s ((s_2, t), []))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cond:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\forall \text{sop. e } \neq \text{Tmp sop } \implies \varnothing \vdash (\text{Cond e s_1 s_2, t}) \rightarrow_s)</td>
</tr>
<tr>
<td>(((\text{Cond (Tmp (eval-expr t e)) s_1 s_2, t + } \text{used-tmps e}, \text{issue-expr t e}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CondTrue:</th>
</tr>
</thead>
<tbody>
<tr>
<td>([D \subseteq \text{dom } \varnothing; \text{isTrue (e } \varnothing)] \implies \varnothing \vdash (\text{Cond (Tmp (D,e)) s_1 s_2, t}) \rightarrow_s ((s_1, t),[]))</td>
</tr>
</tbody>
</table>
CondFalse:
[D ⊆ dom Ᵽ; ¬ isTrue (e Ᵽ)]
[\[D ⊆ \text{dom} \ θ; \neg \text{isTrue} (e \ θ)\]]
\[\Rightarrow \ θ \vdash (\text{Cond} (\text{Tmp} (D, e)) s_1 s_2, t) \rightarrow_s ((s_2, t), [])\]

While:
\[\ θ \vdash (\text{While} \ e \ s, t) \rightarrow_s ((\text{Cond} e (\text{Seq} s (\text{While} e \ s)) \text{Skip}, t), [])\]

SGhost:
\[\ θ \vdash (\text{SGhost} \ A \ L \ R \ W, t) \rightarrow_s ((\text{Skip}, t), [\text{Ghost} (A \ θ) (L \ θ) (R \ θ) (W \ θ)])\]

SFence:
\[\ θ \vdash (\text{SFence}, t) \rightarrow_s ((\text{Skip}, t), [\text{Fence}])\]

\textbf{inductive-cases} stmt-step-cases [cases set]:
\[\ θ \vdash (\text{Skip}, t) \rightarrow_s c\]
\[\ θ \vdash (\text{Assign} \ \text{volatile} \ a \ e \ A \ L \ R \ W, t) \rightarrow_s c\]
\[\ θ \vdash (\text{CAS} \ a \ c_e \ s_e \ A \ L \ R \ W, t) \rightarrow_s c\]
\[\ θ \vdash (\text{Seq} s_1 s_2, t) \rightarrow_s c\]
\[\ θ \vdash (\text{While} \ e \ s_1 s_2, t) \rightarrow_s c\]
\[\ θ \vdash (\text{SGhost} \ A \ L \ R \ W, t) \rightarrow_s c\]
\[\ θ \vdash (\text{SFence}, t) \rightarrow_s c\]

\textbf{lemma} valid-sops-expr-mono:
\[\forall t \ t'. \text{valid-sops-expr} \ t \ e \Rightarrow t \leq t' \Rightarrow \text{valid-sops-expr} \ t' \ e\]
\[\text{by (induct e) auto}\]

\textbf{lemma} valid-sops-stmt-mono:
\[\forall t \ t'. \text{valid-sops-stmt} \ t \ s \Rightarrow t \leq t' \Rightarrow \text{valid-sops-stmt} \ t' \ s\]
\[\text{by (induct s) (auto intro: valid-sops-expr-mono)}\]

\textbf{lemma} valid-sops-expr-valid-sop:
\[\forall t. \text{valid-sops-expr} \ t \ e \Rightarrow \text{valid-sop} (\text{eval-expr} \ t \ e)\]
\textbf{proof} (induct e)
\[\text{case} (\text{Unop} \ f \ e)\]
\[\text{then obtain} \ \text{valid-sops-expr} \ t \ e\]
\[\text{by simp}\]
\[\text{from} \ \text{Unop.hyps} [\text{OF this}]\]
\[\text{have} \ vs: \text{valid-sop} (\text{eval-expr} \ t \ e)\]
\[\text{by simp}\]
\[\text{obtain} \ D \ g \ \text{where} \ \text{eval-e: eval-expr} \ t \ e = (D, g)\]
\[\text{by (cases eval-expr} \ t \ e)\]
\[\text{interpret} \ \text{valid-sop} (D, g)\]
\[\text{using} \ vs \ \text{eval-e}\]
\[\text{by simp}\]

\textbf{show} \ ?\text{case}
apply (clarsimp simp add: Let-def valid-sop-def eval-e)
apply (drule valid-sop [OF refl])
apply simp
done

next

case (Binop f e1 e2)
then obtain v1: valid-sops-expr t e1 and v2: valid-sops-expr t e2
by simp

with Binop.hyps (1) [of t] Binop.hyps (2) [of (t + used-tmpts e1)]
valid-sops-expr-mono [OF v2, of (t + used-tmpts e1)]

obtain vs1: valid-sop (eval-expr t e1) and vs2: valid-sop (eval-expr (t + used-tmpts e1) e2)
by auto

obtain D1 g1 where eval-e1: eval-expr t e1 = (D1,g1)
by (cases eval-expr t e1)

obtain D2 g2 where eval-e2: eval-expr (t + used-tmpts e1) e2 = (D2,g2)
by (cases eval-expr (t + used-tmpts e1) e2)

interpret vs1: valid-sop (D1,g1)
using vs1 eval-e1 by auto

interpret vs2: valid-sop (D2,g2)
using vs2 eval-e2 by auto

{ fix θ :: nat⇒val option

assume D1: D1 ⊆ dom θ

assume D2: D2 ⊆ dom θ

have f (g1 θ) (g2 θ) = f (g1 (θ ↣ (D1 ∪ D2))) (g2 (θ ↣ (D1 ∪ D2)))

proof
from vs1.valid-sop [OF refl D1]

have g1 θ = g1 (θ ↣ D1).

also
from D1 have D1': D1' ⊆ dom (θ ↣ (D1 ∪ D2))

by auto

have θ ↣ (D1 ∪ D2) |' D1 = θ ↣ D1

apply (rule ext)

apply (auto simp add: restrict-map-def)
done

with vs1.valid-sop [OF refl D1']

have g1 (θ ↣ D1) = g1 (θ ↣ (D1 ∪ D2))

by auto

finally have g1: g1 (θ ↣ (D1 ∪ D2)) = g1 θ

by simp

from vs2.valid-sop [OF refl D2]

have g2 θ = g2 (θ ↣ D2).

also
from D2 have D2': D2' ⊆ dom (θ ↣ (D1 ∪ D2))

by auto

have θ ↣ (D1 ∪ D2) |' D2 = θ ↣ D2

apply (rule ext)

apply (auto simp add: restrict-map-def)
done

with vs2.valid-sop [OF refl D2]\]

have g2 (θ | D2) = g2 (θ | (D1 ∪ D2))

by auto

finally have g2: g2 (θ | (D1 ∪ D2)) = g2 θ

by simp

from g1 g2 show ?thesis by simp

qed

note lem=this

show ?case

apply (clarsimp simp add: Let-def valid-sop-def eval-e1 eval-e2)

apply (rule lem)

by auto

qed (auto simp add: valid-sop-def)

lemma valid-sops-expr-eval-expr-in-range:

∀t. valid-sops-expr t e =⇒ ∀t′ ∈ fst (eval-expr t e). t′ < t + used-tmps e

proof (induct e)

case (Unop f e)

thus ?case

apply (cases eval-expr t e)

apply auto

done

next

case (Binop f e1 e2)

then obtain v1: valid-sops-expr t e1 and v2: valid-sops-expr t e2

by simp

from valid-sops-expr-mono [OF v2]

have v2': valid-sops-expr (t + used-tmps e1) e2

by auto

from Binop.hyps (1) [OF v1] Binop.hyps (2) [OF v2']

show ?case

apply (cases eval-expr t e1)

apply (cases eval-expr (t + used-tmps e1) e2)

apply auto

done

qed auto

lemma stmt-step-tmps-count-mono:

assumes step: θ ⊢ (s, t) →ₜ ((s', t'), is)

shows t ≤ t'

using step

by (induct x==(s, t) y==(s', t'), is) arbitrary: s t s' t' is rule: stmt-step.induct) force+
lemma valid-sops-stmt-invariant:
  assumes step: \( \theta \vdash (s,t) \rightarrow_{s} ((s',t'),is) \)
  shows valid-sops-stmt t s \implies valid-sops-stmt t' s'
using step
proof (induct x===(s,t) y===(s',t'),is) arbitrary: s t s' t' is rule: stmt-step.induct)
case AssignAddr thus ?case by (force simp add: valid-sops-expr-valid-sop intro: valid-sops-stmt-mono valid-sops-expr-mono
  dest: valid-sops-expr-eval-expr-in-range)
next
case Assign thus ?case by simp
next
case CASAddr thus ?case by (force simp add: valid-sops-expr-valid-sop intro: valid-sops-stmt-mono valid-sops-expr-mono
  dest: valid-sops-expr-eval-expr-in-range)
next
case CASComp thus ?case by (force simp add: valid-sops-expr-valid-sop intro: valid-sops-stmt-mono valid-sops-expr-mono
  dest: valid-sops-expr-eval-expr-in-range)
next
case CAS thus ?case by simp
next
case Seq thus ?case by (force intro: valid-sops-stmt-mono dest: stmt-step-tmps-count-mono)
next
case SeqSkip thus ?case by auto
next
case Cond thus ?case by (fastforce simp add: valid-sops-expr-valid-sop intro: valid-sops-stmt-mono
  dest: valid-sops-expr-eval-expr-in-range)
next
case CondTrue thus ?case by force
next
case CondFalse thus ?case by force
next
case While thus ?case by auto
next
case SGhost thus ?case by simp
next
case SFence thus ?case by simp
qed

lemma map-le-restrict-map-eq: \( m_1 \subseteq_{m} m_2 \implies D \subseteq \text{dom } m_1 \implies m_2 \mid_{D} m_1 \mid_{D} \)
apply (rule ext)
apply (force simp add: restrict-map-def map-le-def)
done
lemma sbh-step-preserves-load-tmps-bound:
assumes step: (is, O, D, \theta, sb, S, m) \rightarrow_{sbh} (is', O', D', \theta', sb', S', m')
assumes less: \forall i \in load-tmps is'. i < n
shows \forall i \in load-tmps is'. i < n
using step less
by cases auto

lemma sbh-step-preserves-read-tmps-bound:
assumes step: (is, O, D, \theta, sb, S, m) \rightarrow_{sbh} (is', O', D', \theta', sb', S', m')
assumes less-is: \forall i \in load-tmps is. i < n
assumes less-sb: \forall i \in read-tmps sb, i < n
shows \forall i \in read-tmps sb'. i < n
using step less-is less-sb
by cases (auto simp add: read-tmps-append)

lemma sbh-step-preserves-tmps-bound:
assumes step: (is, \theta, sb, m, D, O, S) \rightarrow_{sbh} (is', \theta', sb', m', D', O', S')
assumes less-dom: \forall i \in dom \theta, i < n
assumes less-is: \forall i \in load-tmps is. i < n
shows \forall i \in dom \theta', i < n
using step less-dom less-is
by cases (auto simp add: read-tmps-append)

lemma flush-step-preserves-read-tmps:
assumes step: (m, sb, O) \rightarrow_f (m', sb', O')
assumes less-sb: \forall i \in read-tmps sb, i < n
shows \forall i \in read-tmps sb', i < n
using step less-sb
by cases (auto simp add: read-tmps-append)

lemma flush-step-preserves-write-sops:
assumes step: (m, sb, O) \rightarrow_f (m', sb', O')
assumes less-sb: \forall i \in \bigcup (fst ' write-sops sb), i < t
shows \forall i \in \bigcup (fst ' write-sops sb'), i < t
using step less-sb
by cases (auto simp add: read-tmps-append)

lemma issue-expr-load-tmps-range:
\forall t. load-tmps (issue-expr t e) = \{ i. t \leq i \land i < t + used-tmps e \}
apply (induct e)
apply (force simp add: load-tmps-append)+
done

lemma issue-expr-load-tmps-range:
\forall t. \forall i \in load-tmps (issue-expr t e), t \leq i \land i < t + (used-tmps e)
by (auto simp add: issue-expr-load-tmps-range)
lemma stmt-step-load-tmps-range':
assumes step: \( \emptyset \vdash (s, t) \rightarrow s ((s', t'), is) \)
shows load-tmps is = \{ i. \ t \leq i \land i < t' \} 
using step
apply (induct x==(s,t) y==((s',t'),is) arbitrary: s t s' t' is rule: stmt-step.induct)
apply (force simp add: load-tmps-append simp add: issue-expr-load-tmps-range')+
done

lemma stmt-step-load-tmps-range:
assumes step: \( \emptyset \vdash (s, t) \rightarrow s ((s', t'), is) \)
shows \( \forall i \in \text{load-tmps is. } t \leq i \land i < t' \) 
using stmt-step-load-tmps-range' [OF step]
by auto

lemma distinct-load-tmps-issue-expr: \( \forall t. \text{distinct-load-tmps (issue-expr t e)} \)
apply (induct e)
apply (auto simp add: distinct-load-tmps-append dest!: issue-expr-load-tmps-range [rule-format])
done

lemma max-used-load-tmps: \( t + \text{used-tmps e} \notin \text{load-tmps (issue-expr t e)} \)
proof
from issue-expr-load-tmps-range [rule-format, of t+used-tmps e]
show \( \text{thesis} \)
by auto
qed

lemma stmt-step-distinct-load-tmps:
assumes step: \( \emptyset \vdash (s, t) \rightarrow s ((s', t'), is) \)
shows distinct-load-tmps is 
using step
apply (induct x==(s,t) y==((s',t'),is) arbitrary: s t s' t' is rule: stmt-step.induct)
apply (force simp add: distinct-load-tmps-append distinct-load-tmps-issue-expr max-used-load-tmps)+
done

lemma store-sops-issue-expr [simp]: \( \forall t. \text{store-sops (issue-expr t e)} = \{ \} \)
apply (induct e)
apply (auto simp add: store-sops-append)
done

lemma stmt-step-data-store-sops-range:
assumes step: \( \emptyset \vdash (s, t) \rightarrow s ((s', t'), is) \)
assumes valid: valid-sops-stmt t s
shows \( \forall (D,f) \in \text{store-sops is. } \forall i \in D. \ i < t' \) 
using step valid
proof (induct x==(s,t) y==((s',t'),is) arbitrary: s t s' t' is rule: stmt-step.induct)
case AssignAddr
  thus ?case
    by auto
next
case (Assign D volatile a e)
  thus ?case
    apply (cases eval-expr t e)
    apply (auto simp add: store-sops-append intro: valid-sops-expr-eval-expr-in-range [rule-format])
  done
next
case CASAddr
  thus ?case
    by auto
next
case CASComp
  thus ?case
    by auto
next
case (CAS - - D f a A L R)
  thus ?case
    by (fastforce simp add: store-sops-append dest: valid-sops-expr-eval-expr-in-range [rule-format])
next
case Seq
  thus ?case
    by (force intro: valid-sops-stmt-mono )
next
case SeqSkip thus ?case by simp
next
case Cond thus ?case
  by auto
next
case CondTrue thus ?case by auto
next
case CondFalse thus ?case by auto
next
case While thus ?case by auto
next
case SGhost thus ?case by auto
next
case SFence thus ?case by auto
qed

lemma sbh-step-distinct-load-tmps-prog-step:
  assumes step: \( \theta \vdash (s,t) \rightarrow (s',t'),\is' \) 
  assumes load-tmps-le: \( \forall i \in \text{load-tmps is}. \ i < t \) 
  assumes read-tmps-le: \( \forall i \in \text{read-tmps sb}. \ i < t \) 
  shows distinct-load-tmps is' \wedge (\text{load-tmps is'} \cap \text{load-tmps is} = \{\}) \wedge 
    (\text{load-tmps is'} \cap \text{read-tmps sb}) = \{\} 

733
proof
load-tmps-le read-tmps-le
show ?thesis
by force
qed

lemma data-dependency-consistent-instrs-issue-expr:
\( \forall t T. \text{data-dependency-consistent-instrs } T \ (\text{issue-expr } t e) \)
apply (induct e)
apply (auto simp add: data-dependency-consistent-instrs-append
dest!: issue-expr-load-tmps-range [rule-format])
done

lemma dom-eval-expr:
\( \forall t. \{ \text{valid-sops-expr } t e; x \in \text{fst (eval-expr } t e) \} \Longrightarrow x \in \{i. i < t \} \cup \text{load-tmps (issue-expr } t e) \)
proof (induct e)
case Const thus ?case by simp
next
case Mem thus ?case by simp
next
case Tmp thus ?case by simp
next
case (Unop f e)
thus ?case
by (cases eval-expr t e) auto
next
case (Binop f e1 e2)
then obtain valid1: valid-sops-expr t e1 and valid2: valid-sops-expr t e2
by auto
from valid-sops-expr-mono [OF valid2] have valid2': valid-sops-expr (t+used-tmps e1) e2
by auto
from Binop.hyps (1) [OF valid1] Binop.hyps (2) [OF valid2'] Binop.prems
show ?case
apply (case-tac eval-expr t e1)
apply (case-tac eval-expr (t+used-tmps e1) e2)
apply (auto simp add: load-tmps-append issue-expr-load-tmps-range')
done
qed

lemma Cond-not-s1: s1 \neq \text{Cond } e \ s1 \ s2
by (induct s1) auto

lemma Cond-not-s2: s2 \neq \text{Cond } e \ s1 \ s2
by (induct \(s_2\)) auto

**lemma** Seq-not-s\(_1\): \(s_1 \neq \text{Seq } s_1 \; s_2\)
by (induct \(s_1\)) auto

**lemma** Seq-not-s\(_2\): \(s_2 \neq \text{Seq } s_1 \; s_2\)
by (induct \(s_2\)) auto

**lemma** prog-step-progress:
**assumes** step: \(\theta \vdash (s, t) \rightarrow_s ((s', t'), \text{is})\)
**shows** \((s', t') \neq (s, t) \lor \text{is} \neq []\)
**using** step
**proof** (induct \(x = (s, t) \; y = ((s', t'), \text{is})\) arbitrary: \(s \; t \; s' \; t'\) is rule: stmt-step.induct)
\begin{itemize}
\item **case** (AssignAddr \(a - - - - - - t\)) **thus** ?case
  by (cases eval-expr \(t \; a\)) auto
\item **next**
  case Assign **thus** ?case by auto
\item **next**
  case (CASAddr \(a - - - - - - t\)) **thus** ?case by (cases eval-expr \(t \; a\)) auto
\item **next**
  case (CASComp \(c_e - - - - - - t\)) **thus** ?case by (cases eval-expr \(t \; c_e\)) auto
\item **next**
  case CAS **thus** ?case by auto
\item **next**
  case (Cond \(e - - t\)) **thus** ?case by (cases eval-expr \(t \; e\)) auto
\item **next**
  case CondTrue **thus** ?case **using** Cond-not-s\(_1\) by auto
\item **next**
  case CondFalse **thus** ?case **using** Cond-not-s\(_2\) by auto
\item **next**
  case Seq **thus** ?case by force
\item **next**
  case SeqSkip **thus** ?case **using** Seq-not-s\(_2\) by auto
\item **next**
  case While **thus** ?case by auto
\item **next**
  case SGhost **thus** ?case by auto
\item **next**
  case SFence **thus** ?case by auto
\end{itemize}
qed

**lemma** stmt-step-data-dependency-consistent-instrs:
**assumes** step: \(\theta \vdash (s, t) \rightarrow_s ((s', t'), \text{is})\)
**assumes** valid: valid-sops-stmt \(t \; s\)
**shows** data-dependency-consistent-instrs \(\{i. \; i < t\}\) is
**using** step valid
**proof** (induct \(x = (s, t) \; y = ((s', t'), \text{is})\) arbitrary: \(s \; t \; s' \; t'\) is \(T\) rule: stmt-step.induct)
\begin{itemize}
\item **case** AssignAddr
  **thus** ?case
  by (fastforce simp add: simp add: data-dependency-consistent-instrs-append)
\end{itemize}
735
data-dependency-consistent-instrs-issue-expr load-tmps-append
dest: dom-eval-expr)

next
  case Assign
  thus ?case
    by (fastforce simp add: simp add: data-dependency-consistent-instrs-append
data-dependency-consistent-instrs-issue-expr load-tmps-append
dest: dom-eval-expr)

next
  case CASAddr
  thus ?case
    by (fastforce simp add: simp add: data-dependency-consistent-instrs-append
data-dependency-consistent-instrs-issue-expr load-tmps-append
dest: dom-eval-expr)

next
  case CASComp
  thus ?case
    by (fastforce simp add: simp add: data-dependency-consistent-instrs-append
data-dependency-consistent-instrs-issue-expr load-tmps-append
dest: dom-eval-expr)

next
  case CAS
  thus ?case
    by (fastforce simp add: simp add: data-dependency-consistent-instrs-append
data-dependency-consistent-instrs-issue-expr load-tmps-append
dest: dom-eval-expr)

next
  case Seq
  thus ?case
    by (fastforce simp add: simp add: data-dependency-consistent-instrs-append)

next
  case SeqSkip thus ?case by auto

next
  case Cond
  thus ?case
    by (fastforce simp add: simp add: data-dependency-consistent-instrs-append
data-dependency-consistent-instrs-issue-expr load-tmps-append
dest: dom-eval-expr)

next
  case CondTrue thus ?case by auto

next
  case CondFalse thus ?case by auto

next
  case While
  thus ?case by auto

next
  case SGhost thus ?case by auto

next
  case SFence thus ?case by auto

qed
lemma sbh-valid-data-dependency-prog-step:
assumes step: \( \theta \vdash (s, t) \rightarrow_s ((s', t'), is') \)
assumes store-sops-le: \( \forall i \in \bigcup (f_{s'} \text{ store-sops is}). \ i < t \)
assumes write-sops-le: \( \forall i \in \bigcup (f_{s'} \text{ write-sops sb}). \ i < t \)
assumes valid: valid-sops-stmt t s
shows data-dependency-consistent-instrs \( \{ i. \ i < t \} \) is' \^ 
\begin{align*}
\text{load-tmps is'} & \cap \bigcup (f_{s'} \text{ store-sops is}) = \{} \wedge \\
\text{load-tmps is'} & \cap \bigcup (f_{s'} \text{ write-sops sb}) = \{}
\end{align*}
proof –
from stmt-step-data-dependency-consistent-instrs [OF step valid]
stmt-step-load-tmps-range [OF step]
store-sops-le write-sops-le
show \(?\)thesis
by fastforce
qed

lemma sbh-load-tmps-fresh-prog-step:
assumes step: \( \theta \vdash (s, t) \rightarrow_s ((s', t'), is') \)
assumes tmps-le: \( \forall i \in \text{dom } \theta. \ i < t \)
shows load-tmps is' \cap \text{dom } \theta = \{}
proof –
from stmt-step-load-tmps-range [OF step] tmps-le
show \(?\)thesis
apply auto
subgoal for x
apply (drule-tac x=x in bspec )
apply assumption
apply (drule-tac x=x in bspec )
apply fastforce
apply simp
done
done
qed

lemma sbh-valid-sops-prog-step:
assumes step: \( \theta \vdash (s, t) \rightarrow_s ((s', t'), is) \)
assumes valid: valid-sops-stmt t s
shows \( \forall sop \in \text{store-sops is}. \ \text{valid-sop sop} \)
using step valid
proof (induct x==(s, t) \ y==((s', t'),is) arbitrary: s t s' t' is rule: stmt-step.induct)
case AssignAddr
thus \(?\)case by auto
next
case Assign
thus \(?\)case
by (auto simp add: store-sops-append valid-sops-expr-valid-sop)
next

737
primrec prog-configs:: `'a memref list ⇒ `'a set  

where  

|prog-configs || = {}  
|prog-configs (x#xs) = (case x of  
  Prog sb p p' is ⇒ \{p,p'\} ∪ prog-configs xs  
| - ⇒ prog-configs xs)  

lemma prog-configs-append: ∀ys. prog-configs (xs@ys) = prog-configs xs ∪ prog-configs ys  
by (induct xs) (auto split: memref.splits)  

lemma prog-configs-in1: Prog sb p1 p2 is ∈ set xs ⇒ p1 ∈ prog-configs xs  
by (induct xs) (auto split: memref.splits)  

lemma prog-configs-in2: Prog sb p1 p2 is ∈ set xs ⇒ p2 ∈ prog-configs xs  
by (induct xs) (auto split: memref.splits)  

lemma prog-configs-mono: ∀ys. set xs ⊆ set ys ⇒ prog-configs xs ⊆ prog-configs ys  
by (induct xs) (auto split: memref.splits simp add: prog-configs-append)  

prog-configs-in1 prog-configs-in2

738
locale separated-tmps =
fixes ts
assumes valid-sops-stmt: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O)]
  \implies valid-sops-stmt t s
assumes valid-sops-stmt-sb: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O); (s',t') ∈ prog-configs sb]
  \implies valid-sops-stmt t' s'
assumes load-tmps-le: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O)]
  \implies \forall i ∈ load-tmps is. i < t
assumes read-tmps-le: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O)]
  \implies \forall i ∈ read-tmps sb. i < t
assumes store-sops-le: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O)]
  \implies \forall i ∈ \bigcup (fst ' store-sops is). i < t
assumes write-sops-le: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O)]
  \implies \forall i ∈ \bigcup (fst ' write-sops sb). i < t
assumes tmps-le: [i < length ts; ts!i = ((s,t),is,θ,sb,D,O)]
  \implies dom θ ∪ load-tmps is = \{i. i < t\}

lemma (in separated-tmps)
tmps-le':
assumes i-bound: i < length ts
assumes ts-i: ts!i = ((s,t),is,θ,sb,D,O)
shows \forall i ∈ dom θ. i < t
using tmps-le [OF i-bound ts-i] by auto

lemma (in separated-tmps) separated-tmps-nth-update:
[i < length ts; valid-sops-stmt t s; ∀(s',t') ∈ prog-configs sb. valid-sops-stmt t' s';
  \forall i ∈ load-tmps is. i < t; \forall i ∈ read-tmps sb. i < t;
  \forall i ∈ \bigcup (fst ' store-sops is). i < t; \forall i ∈ \bigcup (fst ' write-sops sb). i < t; dom θ ∪ load-tmps
  is = \{i. i < t\}]  \implies
\text{separated-tmps (ts[i:=((s,t),is,θ,sb,D,O)])}
apply (unfold-locales)
apply (force intro: valid-sops-stmt simp add: nth-list-update split: if-split-asm)
apply (fastforce intro: valid-sops-stmt-sb simp add: nth-list-update-split: if-split-asm)
apply (fastforce intro: load-tmps-le [rule-format] simp add: nth-list-update-split: if-split-asm)
apply (fastforce intro: read-tmps-le [rule-format] simp add: nth-list-update-split: if-split-asm)
apply (fastforce intro: store-sops-le [rule-format] simp add: nth-list-update-split: if-split-asm)
apply (fastforce intro: write-sops-le [rule-format] simp add: nth-list-update-split: if-split-asm)
apply (fastforce dest: tmps-le [rule-format] simp add: nth-list-update-split: if-split-asm)
done
lemma hd-prog-app-in-first: \( \forall ys. \text{Prog}_{sb} p p' \text{ is } \in \text{set} \; xs \implies \text{hd-prog} q (xs \circ ys) = \text{hd-prog} q xs \)
by (induct xs) (auto split: memref.splits)

lemma hd-prog-app-in-eq: \( \forall ys. \text{Prog}_{sb} p p' \text{ is } \in \text{set} \; xs \implies \text{hd-prog} q xs = \text{hd-prog} x xs \)
by (induct xs) (auto split: memref.splits)

lemma hd-prog-app-notin-first: \( \forall ys. \forall p p' \text{ is } \notin \text{set} \; xs \implies \text{hd-prog} q (xs \circ ys) = \text{hd-prog} q ys \)
by (induct xs) (auto split: memref.splits)

lemma union-eq-subsetD: \( A \cup B = C \implies A \cup B \subseteq C \land C \subseteq A \cup B \)
by auto

lemma prog-step-preserves-separated-tmps:
assumes i-bound: \( i < \text{length} \; ts \)
assumes ts-i: \( \text{ts!i} = (p, \theta, \text{sb}, D, O) \)
assumes prog-step: \( \theta \vdash p \rightarrow s (p', \theta, \text{sb}, D, O) \)
assumes sep: separated-tmps ts
shows separated-tmps (ts \([i := ((s', t'), \text{is@is'}, \theta, \text{sb}@[\text{Prog}_{sb} (s, t) (s', t') \text{ is'}], D, O)])
proof –
obtain \( s t \) where \( p = (s, t) \) by (cases p)
obtain \( s' t' \) where \( p' = (s', t') \) by (cases p')
note ts-i = ts-i [simplified p]
note step = prog-step [simplified p p']
interpret separated-tmps ts by fact
have separated-tmps (ts[i := ((s', t'), \text{is@is'}, \theta, \text{sb}@[\text{Prog}_{sb} (s, t) (s', t') \text{ is'}], D, O)])
proof (rule separated-tmps-nth-update [OF i-bound])
stmt-step-tmps-count-mono [OF step]
show \( \forall i \in \text{load-tmps} (\text{is@is'}). i < t' \)
by (auto simp add: load-tmps-append)
next
show \( \forall i \in \text{read-tmps} (\text{sb}@[\text{Prog}_{sb} (s, t) (s', t') \text{ is'}]). i < t' \)
by (auto simp add: read-tmps-append)
next
store-sops-le [OF i-bound ts-i] valid-sops-stmt [OF i-bound ts-i]
show \( \forall i \in \bigcup (\text{fst ' store-sops} (\text{is@is'})). i < t' \)
by (fastforce simp add: store-sops-append)
next
show \( \forall i \in \bigcup (\text{fst ' write-sops} (\text{sb}@[\text{Prog}_{sb} (s, t) (s', t') \text{ is'}]). i < t' \)
by (fastforce simp add: write-sops-append)
next
from tmps-le [OF i-bound ts-i]

740
have dom θ ∪ load-tmps is = {i. i < t} by simp
show dom θ ∪ load-tmps (is @ is') = {i. i < t'}
  apply (clarsimp simp add: load-tmps-append)
  apply rule
  apply (drule union-eq-subsetD)
  apply fastforce
  apply clarsimp
  subgoal for x
  apply (case-tac t ≤ x)
  apply simp
  apply (subgoal-tac x < t)
  apply fastforce
  apply fastforce
  done
  done
next
  from valid-sops-stmt-invariant [OF prog-step [simplified p p'] valid-sops-stmt [OF i-bound ts-i]]
  show valid-sops-stmt t' s'.
next
  show ∀(s', t')∈prog-configs (sb @ [Prog sb (s, t) (s', t') is']).
      valid-sops-stmt t' s'
  proof –
  |
  { fix s₁ t₁
    assume cfgs: (s₁, t₁) ∈ prog-configs (sb @ [Prog sb (s, t) (s', t') is'])
    have valid-sops-stmt t₁ s₁
      proof –
      |
      from valid-sops-stmt [OF i-bound ts-i]
      have valid-sops-stmt t s.
      moreover
      from valid-sops-stmt-invariant [OF prog-step [simplified p p'] valid-sops-stmt [OF i-bound ts-i]]
      have valid-sops-stmt t' s'.
      moreover
      note valid-sops-stmt-sb [OF i-bound ts-i]
      ultimately
      show ?thesis
      using cfgs
      by (auto simp add: prog-configs-append)
  qed
  |
  thus ?thesis
by auto
  qed
  qed
then
  show ?thesis
  by (simp add: p p')
lemma flush-step-sb-subset:
  assumes step: \((m, sb, O) \rightarrow_I (m', sb', O')\)
  shows set sb' \subseteq set sb
using step
apply (induct c1==(m, sb, O) c2==(m', sb', O') arbitrary: m sb O acq m' sb' O' acq
rule: flush-step.induct)
apply auto
done

lemma flush-step-preserves-separated-tmps:
  assumes i-bound: \(i < \text{length ts}\)
  assumes ts-i: \(ts!i = (p, is, \emptyset, sb, D, O, R)\)
  assumes flush-step: \((m, sb, O, R, S) \rightarrow_I (m', sb', O', R', S')\)
  assumes sep: separated-tmps ts
  shows separated-tmps (ts \([i:=((p, is, \emptyset, sb', D, O', R')])\])
proof −
  obtain s t where p: p=(s,t) by (cases p)
  note ts-i = ts-i [simplified p]
  interpret separated-tmps ts by fact
  have separated-tmps (ts \([i:=((s,t), is, \emptyset, sb', D, O', R')])\))
  proof (rule separated-tmps-nth-update [OF i-bound])
    from load-tmps-le [OF i-bound ts-i]
    show \(\forall i \in \text{load-tmps is}. i < t.\)
  next
    show \(\forall i \in \text{read-tmps sb'}. i < t.\)
  next
    from store-sops-le [OF i-bound ts-i]
    show \(\forall i \in \bigcup (\text{fst ' store-sops is}). i < t.\)
  next
    from flush-step-preserves-write-sops [OF flush-step write-sops-le [OF i-bound ts-i]]
    show \(\forall i \in \bigcup (\text{fst ' write-sops sb'}). i < t.\)
  next
    from tmps-le [OF i-bound ts-i]
    show dom \(\emptyset \cup \text{load-tmps is} = \{i. i < t\}\)
      by auto
  next
    from valid-sops-stmt [OF i-bound ts-i]
    show valid-sops-stmt t s.
  next
    from valid-sops-stmt-sb [OF i-bound ts-i] flush-step-sb-subset [OF flush-step]
    show \(\forall (s', t') \in \text{prog-configs sb'}. valid-sops-stmt t' s'\)
      by (auto dest!: prog-configs-mono)
qed
then
show \(?\text{thesis}\)
  by (simp add: p)
qed
lemma sbh-step-preserves-store-sops-bound:
assumes step: (is, θ, sb, m, D, O, R, S) →_{sbh} (is’, θ’, sb’, m’, D’, O’, R’, S’)
assumes store-sops-le: ∀ i ∈ ∪ (fst ' store-sops is). i < t
shows ∀ i ∈ ∪ (fst ' store-sops is’). i < t
using step store-sops-le
by cases auto

lemma sbh-step-preserves-write-sops-bound:
assumes step: (is, θ, sb, m, D, O, R, S) →_{sbh} (is’, θ’, sb’, m’, D’, O’, R’, S’)
assumes store-sops-le: ∀ i ∈ ∪ (fst ' store-sops is). i < t
assumes write-sops-le: ∀ i ∈ ∪ (fst ' write-sops sb). i < t
shows ∀ i ∈ ∪ (fst ' write-sops sb’). i < t
using step store-sops-le write-sops-le
by cases (auto simp add: write-sops-append)

lemma sbh-step-prog-configs-eq:
assumes step: (is, θ, sb, m, D, O, R, S) →_{sbh} (is’, θ’, sb’, m’, D’, O’, R’, S’)
shows prog-configs sb’ = prog-configs sb
using step
apply (cases)
apply (auto simp add: prog-configs-append)
done

lemma sbh-step-preserves-tmps-bound:
assumes step: (is, θ, sb, m, D, O, R, S) →_{sbh} (is’, θ’, sb’, m’, D’, O’, R’, S’)
shows dom θ ∪ load-tmps is = dom θ’ ∪ load-tmps is’
using step
apply cases
apply (auto simp add: read-tmps-append)
done

lemma sbh-step-preserves-separated-tmps:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb, D, O, R)
assumes memop-step: (is, θ, sb, m, D, O, R, S) →_{sbh} (is’, θ’, sb’, m’, D’, O’, R’, S’)
assumes instr: separated-tmps ts
shows separated-tmps (ts [i:=<(p, is’, θ’, sb’, D’, O’, R’)>])
proof –
  obtain s t where p: p=(s, t) by (cases p)
  note ts-i = ts-i [simplified p]
  interpret separated-tmps ts by fact
  have separated-tmps (ts [i:=<(s, t), is’, θ’, sb’, D’, O’, R’>])
  proof (rule separated-tmps-nth-update [OF i-bound])
    from sbh-step-preserves-load-tmps-bound [OF memop-step load-tmps-le [OF i-bound ts-i]]
    show ∀ i ∈ load-tmps is’. i < t.
  next

743
from sbh-step-preserves-read-tmps-bound [OF memop-step load-tmps-le [OF i-bound ts-i]]
  read-tmps-le [OF i-bound ts-i]]
show \( \forall i \in \text{read-tmps sb}^\prime \). \( i < t \).
next
from sbh-step-preserves-store-sops-bound [OF memop-step store-sops-le [OF i-bound ts-i]]
show \( \forall i \in \bigcup (\text{fst } \text{store-sops is})^\prime \). \( i < t \).
next
from sbh-step-preserves-write-sops-bound [OF memop-step store-sops-le [OF i-bound ts-i]]
write-sops-le [OF i-bound ts-i]]
show \( \forall i \in \bigcup (\text{fst } \text{write-sops sb})^\prime \). \( i < t \).
next
from sbh-step-preserves-tmps-bound' [OF memop-step] tmps-le [OF i-bound ts-i]
show \( \text{dom } \emptyset^\prime \cup \text{load-tmps is}^\prime = \{ i. \ i < t \} \)
by auto
next
from valid-sops-stmt [OF i-bound ts-i]
show valid-sops-stmt t s.
next
show \( \forall (s^\prime, t^\prime) \in \text{prog-configs sb}^\prime \). valid-sops-stmt t' s'
by auto
qed
then show ?thesis
  by (simp add: p)
qed

definition valid-pimp ts ≡ separated-tmps ts

lemma prog-step-preserves-valid:
  assumes i-bound: \( i < \text{length ts} \)
  assumes ts-i: ts!i = (p, is, \emptyset, sb::stmt-config store-buffer, D, O, R)
  assumes prog-step: \( \emptyset \vdash p \rightarrow_s (p', \text{is}^\prime) \)
  assumes valid: valid-pimp ts
  shows valid-pimp (ts[i:=(p', is@is', \emptyset, sb@sb)]\[Prog sb p p' is'\], D, O, R])
using prog-step-preserves-separated-tmps [OF i-bound ts-i prog-step] valid
by (auto simp add: valid-pimp-def)

lemma flush-step-preserves-valid:
  assumes i-bound: \( i < \text{length ts} \)
  assumes ts-i: ts!i = (p, is, \emptyset, sb::stmt-config store-buffer, D, O, R)
  assumes flush-step: \( (m, sb, O, R, S) \rightarrow_t (m', sb', O', R', S') \)
  assumes valid: valid-pimp ts
  shows valid-pimp (ts[i:=(p, is@is', \emptyset, sb@sb)]\[D, O', R'])
using flush-step-preserves-separated-tmps [OF i-bound ts-i flush-step] valid
by (auto simp add: valid-pimp-def)
lemma sbh-step-preserves-valid:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, 0, sb::stmt-config store-buffer, D, O, R)
assumes memop-step: (is, 0, sb, m, D, O, R, S) \rightarrow_{sbh} (is', 0', sb', m', D', O', R', S')
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:= (p, is', 0', sb', D', O', R')])
using
sbh-step-preserves-separated-tmps [OF i-bound ts-i memop-step] valid
by (auto simp add: valid-pimp-def)

lemma hd-prog-prog-configs: hd-prog p sb = p ∨ hd-prog p sb ∈ prog-configs sb
by (induct sb) (auto split:memref.splits)

interpretation PIMP: xvalid-program-progress stmt-step \(\lambda(s, t).\) valid-sops-stmt t s valid-pimp
proof
fix \(\theta\) p p' is'
assume step: \(\theta \vdash p \rightarrow_s (p', is')\)
obtain s t where p: p = (s, t)
by (cases p)
obtain s' t' where p': p' = (s', t')
by (cases p')
from prog-step-progress [OF step [simplified p p']] show p' \(\neq p \lor is' \neq []\)
by (simp add: p p')
next
fix \(\theta\) p p' is'
assume step: \(\theta \vdash p \rightarrow_s (p', is')\)
and valid-stmt: \(\lambda(s, t).\) valid-sops-stmt t s p
 obtain s t where p: p = (s, t)
 by (cases p)
obtain s' t' where p': p' = (s', t')
by (cases p')
from valid-sops-stmt-invariant [OF step [simplified p p']] valid-stmt [simplified p, simplified]
have valid-sops-stmt t's'.
then show \((\lambda(s, t).\) valid-sops-stmt t s\) p' by (simp add: p')
next
fix i ts p is O R D \(\theta\) sb
assume i-bound: i < length ts
and ts-i: ts ! i = (p, is, \theta, sb::(stmt × nat) memref list, D, O, R)
and valid: valid-pimp ts
from valid have separated-tmps ts
by (simp add: valid-pimp-def)
then interpret separated-tmps ts .
obtain s t where p: p = (s, t)
by (cases p)
from valid-sops-stmt [OF i-bound ts-i [simplified p]]
show \((\lambda(s, t). \text{valid-sops-stmt } t \ s) \ p\)
by (auto simp add: \(p\))

next
fix \(i\) \(ts\) \(p\) is \(\mathcal{O} \mathcal{R} \mathcal{D} \ \emptyset \ \text{sb}\)
assume i-bound: \(i < \text{length ts}\)
and ts-i: \(\text{ts} ! i = (p, \text{is}, \emptyset, \text{sb}:(\text{stmt} \times \text{nat}) \ \text{memref list}, \mathcal{D}, \mathcal{O}, \mathcal{R})\)
and valid: valid-pimp ts
from valid have separated-tmps ts
by (simp add: valid-pimp-def)
then interpret separated-tmps ts .
obtain \(s\) \(t\) where \(p: p = (s, t)\)
by (cases \(p\))
from \(\text{hd-prog-prog-configs [of } p \text{ sb]}\) valid-sops-stmt [OF i-bound ts-i [simplified \(p\)]]
valid-sops-stmt-sb [OF i-bound ts-i [simplified \(p\)]]
show \((\lambda(s, t). \text{valid-sops-stmt } t \ s) \ (\text{hd-prog } p \ \text{sb})\)
by (auto simp add: \(p\))

next
fix \(i\) \(ts\) \(p\) \(\text{is} \ θ \ \text{sb} \ \text{is}'\)
assume i-bound: \(i < \text{length ts}\)
and ts-i: \(\text{ts} ! i = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R})\)
and step: \(\emptyset \vdash p \rightarrow_s (p', \text{is}')\)
and valid: valid-pimp ts
show distinct-load-tmps \(\text{is}'\ ∧ \text{load-tmps is}' \cap \text{load-tmps is} = \{\} ∧ \text{load-tmps is}' \cap \text{read-tmps sb} = \{\}\)
proof –
obtain \(s\) \(t\) where \(p: p = (s, t)\)
by (cases \(p\))
obtain \(s'\) \(t'\) where \(p': p' = (s', t')\)
by (cases \(p'\))
note ts-i = ts-i [simplified \(p\)]
note step = step [simplified \(p\)
\(p\')]
from valid
interpret separated-tmps ts
by (simp add: valid-pimp-def)

from \(\text{sbh-step-distinct-load-tmps-prog-step [of step load-tmps-le [OF i-bound ts-i]
read-tmps-le [OF i-bound ts-i]}\]
show \(?\text{thesis}\).
qed

next
fix \(i\) \(ts\) \(p\) \(\mathcal{O} \mathcal{R} \mathcal{D} \ \emptyset \ \text{sb} \ \text{is}'\)
assume i-bound: \(i < \text{length ts}\)
and ts-i: \(\text{ts} ! i = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R})\)
and step: \(\emptyset \vdash p \rightarrow_s (p', \text{is}')\)
and valid: valid-pimp ts
show data-dependency-consistent-instrs (dom \(\emptyset \cup \text{load-tmps is}\)) \(\text{is}'\ ∧ \text{load-tmps is}' \cap \bigcup (\text{fst ' store-sops is}) = \{\} ∧ \text{load-tmps is}' \cap \bigcup (\text{fst ' write-sops sb}) = \{\}\)
proof –
obtain \(s\) \(t\) where \(p: p = (s, t)\)
by (cases \(p\))
obtain $s' t'$ where $p' = (s' t')$ by (cases $p'$)

note $ts-i = ts-i$ [simplified $p$]

note step = step [simplified $p$ $p'$]

from valid
interpret separated-tmps $ts$
  by (simp add: valid-pimp-def)

from sbh-valid-data-dependency-prog-step [OF step store-sops-le [OF i-bound $ts-i$]]

show ?thesis by auto

qed

next

fix $i$ $ts$ $p$ is $O R D \emptyset sb p'$ $is'$

assume i-bound: $i < \text{length } ts$
  and $ts-i: ts ! i = (p, is, \emptyset, sb, D, O, R)$
  and step: $\emptyset \vdash p \rightarrow s (p', is')$
  and valid: valid-pimp $ts$

show load-tmps is' $\cap \text{dom } \emptyset = \{\}$

proof –
  obtain $s$ $t$ where $p: p = (s, t)$ by (cases $p$)
  obtain $s' t'$ where $p' = (s' t')$ by (cases $p'$)
  note $ts-i = ts-i$ [simplified $p$]
  note step = step [simplified $p$ $p'$]
  from valid
interpret separated-tmps $ts$
  by (simp add: valid-pimp-def)
from sbh-load-tmps-fresh-prog-step [OF step tmps-le' [OF i-bound $ts-i$]]
show ?thesis .

qed

next

fix $\emptyset$ $p$ $p'$ $is$

assume step: $\emptyset \vdash p \rightarrow s (p', is)$
  and valid: $(\lambda(s, t). \text{valid-sops-stmt } t s) p$

show $\forall sop \in \text{store-sops is. valid-sop sop}$

proof –
  obtain $s$ $t$ where $p: p = (s, t)$ by (cases $p$)
  obtain $s' t'$ where $p' = (s' t')$ by (cases $p'$)
  note step = step [simplified $p$ $p'$]
  from sbh-valid-sops-prog-step [OF step valid [simplified $p$, simplified]]
  show ?thesis .

qed

next

fix $i$ $ts$ $p$ is $O R D \emptyset sb p'$ $is'$

assume i-bound: $i < \text{length } ts$
  and $ts-i: ts ! i = (p, is, \emptyset, sb::stmt-config store-buffer, D, O, R)$
  and step: $\emptyset \vdash p \rightarrow s (p', is')$
  and valid: valid-pimp $ts$
from prog-step-preserves-valid [OF i-bound $ts-i$ step valid]
show valid-pimp $(ts[i := (p', is @ is', \emptyset, sb @ [Prog_{sb} p p' is']], D, O, R)] )$. 747
next
fix i ts p is O \ R \ D \ \partial \ sb \ S m m' sb' O' R' S'
assume i-bound: i < length ts
  and ts-i: ts ! i = (p, is, \partial, sb::stmt-config store-buffer, D, O, R)
  and step: (m, sb, O, R, S) \rightarrow_{t} (m', sb', O', R', S')
  and valid: valid-pimp ts
thm flush-step-preserves-valid [OF ]
from flush-step-preserves-valid [OF i-bound ts-i step valid]
show valid-pimp (ts[i := (p, is, \partial, sb', D, O', R')]) .
qed

thm PIMP.concurrent-direct-steps-simulates-store-buffer-history-step
thm PIMP.concurrent-direct-steps-simulates-store-buffer-history-steps
thm PIMP.concurrent-direct-steps-simulates-store-buffer-step

We can instantiate PIMP with the various memory models. interpretation direct:
computation direct-memop-step empty-storebuffer-step stmt-step \lambda p p' is sb. ()
interpretation virtual:
computation virtual-memop-step empty-storebuffer-step stmt-step \lambda p p' is sb. ()
interpretation store-buffer:
computation sb-memop-step store-buffer-step stmt-step \lambda p p' is sb. sb
interpretation store-buffer-history:
computation sbh-memop-step flush-step stmt-step \lambda p p' is sb. sb \@ [Prog_{sb} p p' is].

abbreviation direct-pimp-step::
(stmt-config,unit,BOOL,owns,rels,shared) global-config \Rightarrow
(stmt-config,unit,BOOL,owns,rels,shared) global-config \Rightarrow Boolean (- \Rightarrow_{dp} [60,60] 100)
where
c \Rightarrow_{dp} d \equiv \text{direct.concurrent-step } c \ d

abbreviation direct-pimp-steps::
(stmt-config,unit,BOOL,owns,rels,shared) global-config \Rightarrow
(stmt-config,unit,BOOL,owns,rels,shared) global-config \Rightarrow Boolean (- \Rightarrow_{dp} [60,60] 100)
where
direct-pimp-steps == direct-pimp-step^**

Execution examples lemma Assign-Const-ex:
([(Assign \ True (Tmp \ {{{}, \lambda \theta. a} \ Const c \ (\lambda \theta. A) \ (\lambda \theta. L) \ (\lambda \theta. R) \ (\lambda \theta. W))], ]) \partial(\partial(), D, O, R), m, S) \Rightarrow_{dp}^*
([(Skip,t,c], \partial(), \text{True}, O \cup A - R, \text{Map.empty}), m(a := c), S \oplus_{W} R \oplus_{A} L)
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Program [where i=0])
apply simp
apply simp
apply (rule Assign)
apply simp
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Memop [where i=0])
apply simp
apply simp
apply (rule direct-memop-step.WriteVolatile)
apply simp
done

lemma

apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Memop [where i=0])
apply simp
apply simp
apply (rule Assign)
apply simp
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Memop)
apply simp
apply simp
apply (rule direct-memop-step.Read)
apply simp

apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Memop)
apply simp
apply simp
apply (rule direct-memop-step.Read)
apply simp

apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Memop)
apply simp
apply simp
apply (rule direct-memop-step.WriteVolatile)
apply simp
done

lemma

749
assumes isTrue: isTrue c

shows

\[
((\text{Cond } (\text{Const } c) \ (\text{Assign True } \text{Tmp } \{\}, \lambda \theta. a)) \ (\text{Const } c) \ (\lambda \theta. A) \ (\lambda \theta. L) \ (\lambda \theta. R) \ (\lambda \theta. W)) \ \text{Skip}, t) \ , \ [\, , \theta, (\{D, O, R\}, m, S) \Rightarrow^{*} \ \\
((\text{Skip}, t) \ , \ [\, , \theta, (\{\text{True}, O \cup \ A - R, \text{Map.empty}\}, m(a := c), S \oplus W R \ominus A L) \ \\
\text{apply (rule converse-rtranclp-into-rtranclp)} \ \\
\text{apply (rule direct.Program [where i=0])} \ \\
\text{apply simp} \ \\
\text{apply simp} \ \\
\text{apply (rule Cond)} \ \\
\text{apply simp} \ \\
\text{apply simp} \ \\
\text{apply (rule converse-rtranclp-into-rtranclp)} \ \\
\text{apply (rule direct.Program [where i=0])} \ \\
\text{apply simp} \ \\
\text{apply simp} \ \\
\text{apply (rule CondTrue)} \ \\
\text{apply simp} \ \\
\text{apply (simp add: isTrue)} \ \\
\text{apply simp} \ \\
\text{apply (rule Assign-Const-ex)} \ \\
\text{done} \]
    rev. 29.