A Reduction Theorem for Store Buffers

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

\begin{footnotesize}
\textsuperscript{1} Microsoft Corp., Redmond, WA, USA
\textsuperscript{2} German Research Center for Artificial Intelligence (DFKI) Saarbrücken, Germany
\end{footnotesize}

\texttt{ecohen@amazon.com, norbert.schirmer@web.de}

\textbf{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:

```
T0
x = 1;
r0 = y;
```

```
T1
y = 1;
r1 = x;
```

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 atomic4).

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 paper 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., `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., \( \text{nat} \Rightarrow \text{nat} \) 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 \( \bot \). A base value can be projected with the function the, which is defined by the sole equation \( \text{the } \lfloor x \rfloor = x \). Since HOL is a total logic the term the \( \bot \) 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 \( x # \) the element \( x \) is ‘consed’ to the list \( xs \). With \( xs @ ys \) the 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., \( -A \).

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 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 1: Programming discipline.

<table>
<thead>
<tr>
<th>(a) Access policy</th>
<th>(b) Flushing policy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>shared</td>
<td>shared</td>
</tr>
<tr>
<td>(read-write)</td>
<td>(read-only)</td>
</tr>
<tr>
<td>owned</td>
<td>owned</td>
</tr>
<tr>
<td>owned by other</td>
<td></td>
</tr>
<tr>
<td>vR, vW</td>
<td>vR, R</td>
</tr>
<tr>
<td>vR</td>
<td>unreachable</td>
</tr>
<tr>
<td>volatile, (R)ead, (W)rite</td>
<td>flush (before)</td>
</tr>
<tr>
<td>all reads have to be clean</td>
<td>interlocked as side effect</td>
</tr>
<tr>
<td></td>
<td>vR</td>
</tr>
<tr>
<td></td>
<td>R, vW, W</td>
</tr>
</tbody>
</table>

7
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 \( \text{volatile} \) 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:

\[
\text{valid-sop } sop \equiv \forall D \ f \ \theta. \ sop = (D, f) \land D \subseteq \text{dom } \theta \implies f \ |_{\theta[D]} = f \ |_{\theta[D]}
\]
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, \( \theta \), sb) consisting of the program state p, a memory instruction list is, the map of temporaries \( \theta \) 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 (ts, m) \( \xrightarrow{sb} \) (ts', m') defined in Figure 2.

\[
\begin{align*}
i < |ts| & \quad ts[i] = (p, is, \theta, sb) \quad \Rightarrow \quad p \rightarrow_p (p', is') \\
& \quad (ts, m) \xrightarrow{sb} (ts[i := (p', is', \theta, sb)], m) \\

i < |ts| & \quad ts[i] = (p, is, \theta, sb) \quad (is, \theta, sb, m) \xrightarrow{m} (is', \theta', sb', m') \\
& \quad (ts, m) \xrightarrow{sb} (ts[i := (p, is', \theta', sb')], m') \\

i < |ts| & \quad ts[i] = (p, is, \theta, sb) \quad (m, sb) \rightarrow_{sb} (m', sb') \\
& \quad (ts, m) \xrightarrow{sb} (ts[i := (p, is, \theta, sb')], m')
\end{align*}
\]

Fig. 2: Global transitions of store buffer machine

A transition selects a thread ts[i] = (p, is, \( \theta \), 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 \( \theta \rightarrow_p (p', is') \) takes the temporaries \( \theta \) 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, \( \theta \), sb, m) \( \xrightarrow{m} (is', \theta', sb', m') \) 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 \textproc{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.
\( \text{(m, Write}_{ab} \text{ volatile a sop v A L R W} \# sb) \rightarrow_{sb} (m(a := v), sb) \)

Fig. 3: Store buffer transition

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

Fig. 4: Memory transitions of store buffer machine

The memory transition are defined in Figure 4. With 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 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 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 \((p, \text{is}, \vartheta, D, O)\), 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: 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 safety judgment \(O_i \vdash (\text{is}, \vartheta, m, D, O, S) \sqrt{v}\) in Figure 7, where \(O_i\) is the list of ownership sets obtained from the thread list \(ts\) and \(i\) is the index of the current
\[
\begin{align*}
  i < |ts| & \quad ts[i] = (p, is, \emptyset, D, O) \quad \varnothing \vdash p \rightarrow_p (p', is') \\
  (ts, m, S) & \Rightarrow (ts[i := (p', is', \emptyset, D, O)], m, S)
\end{align*}
\]

\[
\begin{align*}
  i < |ts| & \quad ts[i] = (p, is, \emptyset, D, O) \quad (is, \emptyset, m, D, O, S) \sim_m (is', \emptyset', m', D', O', S') \\
  (ts, m, S) & \Rightarrow (ts[i := (p, is', D', O')], m', S')
\end{align*}
\]

**Fig. 5:** Global transitions of virtual machine

(Read volatile a \( t \# \) is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset(t \mapsto m a), x, m, ghst) \)

(Write False a (D, f) A L R W \# is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset, x, m(a := f \emptyset), ghst) \)

\( ghst = (D, O, S) \quad ghst' = (\text{True}, O \cup A \setminus R, S \oplus W \ R \ominus A \ L) \)

(Write True a (D, f) A L R W \# is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset, x, m(a := f \emptyset), ghst') \)

\( \neg \text{cond} (\emptyset(t \mapsto m a)) \quad ghst = (D, O, S) \quad ghst' = (\text{False}, O, S) \)

(RMW a t (D, f) cond ret A L R W \# is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset(t \mapsto m a), x, m, ghst') \)

\( \text{cond} (\emptyset(t \mapsto m a)) \quad \emptyset' = \emptyset(t \mapsto \text{ret} (m a)) (f (\emptyset(t \mapsto m a))) \)

\( m' = m(a := f (\emptyset(t \mapsto m a))) \quad ghst = (D, O, S) \quad ghst' = (\text{False}, O \cup A \setminus R, S \oplus W \ R \ominus A \ L) \)

(RMW a t (D, f) cond ret A L R W \# is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset', x, m', ghst') \)

\( ghst = (D, O, S) \quad ghst' = (\text{False}, O, S) \)

(Fence \( \# \) is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset, x, m, ghst') \)

\( ghst = (D, O, S) \quad ghst' = (D, O \cup A \setminus R, S \oplus W \ R \ominus A \ L) \)

(Ghost A L R W \# is, \( \emptyset \), x, m, ghst) \( \Rightarrow_m (is, \emptyset, x, m, ghst') \)

**Fig. 6:** Memory transitions of the virtual machine

11
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

\[ a \in \mathcal{O} \lor a \in \text{read-only} \mathcal{S} \lor \text{volatile} \land a \in \text{dom } \mathcal{S} \quad \text{volatile} \rightarrow \neg \mathcal{D} \]

\[ O_s, i \vdash (\text{Read volatile } a \ t \neq \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \checkmark \]

\[ a \in \mathcal{O} \quad a \notin \text{dom } \mathcal{S} \]

\[ O_s, i \vdash (\text{Write False } a (\mathcal{D}, t) \ A L R W \neq \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \checkmark \]

\[ \forall j < |OS|, i \neq j \rightarrow a \notin \text{read-only } \mathcal{S} \]

\[ O_s, i \vdash (\text{Write True } a (\mathcal{D}, t) \ A L R W \neq \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \checkmark \]

\[ \neg \text{cond } (\emptyset(t \rightarrow m a)) \quad a \in \text{dom } \mathcal{S} \cup \mathcal{O} \]

\[ O_s, i \vdash (\text{RMW } a t (\mathcal{D}, t) \text{ cond ret } A L R W \neq \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \checkmark \]

\[ \forall j < |OS|, i \neq j \rightarrow a \notin \text{read-only } \mathcal{S} \]

\[ O_s, i \vdash (\text{FENCE } \neq \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \checkmark \]

\[ A \subseteq \text{dom } \mathcal{S} \cup \mathcal{O} \quad L \subseteq A \quad R \subseteq \mathcal{O} \quad A \cap R = \emptyset \quad \forall j < |OS|, i \neq j \rightarrow A \cap OS[j] = \emptyset \]

\[ O_s, i \vdash (\text{Ghost } A L R W \neq \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \checkmark \]

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 \( R \in \mathcal{O} \cup A \rightarrow R \).
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: \( S \oplus W \)
\( R \ominus A \ L \). The auxiliary ternary operators to augment and subtract addresses from the
sharing map are defined as follows:

\[ S \oplus_W R \equiv \lambda a. \text{ if } a \in R \text{ then } [a \in W] \text{ else } S a \]

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

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 $\text{Ghost A L R W}$ 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 $(\text{ts}_{\text{sb}},
m) \xrightarrow{s} (\text{ts}_{\text{sb}}', m')$ can be simulated by a computation of the virtual machine $(\text{ts}, m, S) \xrightarrow{s} (\text{ts}', m', S')$. 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 $\text{ts}_{\text{sb}} \sim \text{ts}$.
Moreover, the precondition $\text{initial}_v \text{ts} S$ 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 $\text{safe-reach} (\text{ts}, m, S)$ 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).

\begin{align*}
\text{Theorem 1 (Reduction).} \\
(t_{\text{sb}}, m) \xrightarrow{s} (t_{\text{sb}}', m') \land \text{empty-store-buffers} \quad t_{\text{sb}}' \land t_{\text{sb}} \sim \text{ts} \land \text{initial}_v \text{ts} S \land \\
\text{safe-reach} (\text{ts}, m, S) \rightarrow \\
\exists \text{ts}' S'. (ts, m, S) \xrightarrow{s} (ts', m', S') \land t_{\text{sb}}' \sim \text{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 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 intermediate config-
uration 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 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 collap 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
\((i, s, \theta, m, D, O, R, S) \to_m (i', s', \theta', m', D', O', R', S')\) 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:

\[
\text{aug } (\text{dom } S) \ R \ =
\]
(Read volatile a t # is, ∅, m, ghst) \xrightarrow{\lambda \alpha} (is, ∅(t \mapsto m a), m, ghst)

(Write False a (D, f) A L R W # is, ∅, m, ghst) \xrightarrow{\lambda \alpha} (is, ∅, m(a := f ∅), ghst)

ghst = (D, O, R, S) ghst' = (True, O \cup A - R, empty, S \oplus W R \ominus A L)

(Write True a (D, f) A L R W # is, ∅, m, ghst) \xrightarrow{\lambda \alpha} (is, ∅, m(a := f ∅), ghst')

\neg \text{ cond } (∅(t \mapsto m a)) ghst = (D, O, R, S) ghst' = (False, O, empty, S)

(RMW a t (D, f) cond ret A L R W # is, ∅, m, ghst) \xrightarrow{\lambda \alpha} (is, ∅', m', ghst')

(Fence # is, ∅, m, D, O, R, S) \xrightarrow{\lambda \alpha} (is, ∅, m, False, O, empty, S)

ghst = (D, O, R, S) ghst' = (D, O \cup A - R, \text{aug (dom S)} R R, S \oplus W R \ominus A L)

(GHOST A L R W # is, ∅, m, ghst) \xrightarrow{\lambda \alpha} (is, ∅, 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 loose information about a release of an unshared address.

The global transition (ts, m, s) \xrightarrow{\lambda \alpha} (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 Os,Rs,i- (is, ∅, m, D, O, S) \checkmark 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 \notin dom Rs[j]) \checkmark; reading from an address is only allowed if it is not released by another thread while it was unshared (Rs[j] a = [False]).

For the store buffer machine with history information we not only put writes 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

- \text{READ}_{\text{sb}} \text{ volatile a t v, recording a corresponding read from address a which loaded the value v to temporary t, or a}
- \text{WRITE}_{\text{sb}} \text{ volatile a sop v for an outstanding write, where operation sop evaluated to value v, or of the form}
- \text{PROC}_{\text{sb}} p p'is', recording a program transition from p to p' which issued instructions is', or of the form
- \text{GHOST}_{\text{sb}} 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
\( a \in O \lor a \in \text{read-only } S \lor \text{volatile } a \in \text{dom } S \quad \forall j < |O_s|, i \neq j \rightarrow R_{s[j]} a \neq |\text{False}|
\)

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

\[
O_s, R_s, i \vdash (\text{Read volatile } a \ t \ \# \ is, \ \emptyset, m, D, O, S) \checkmark
\]

\[
a \in O \quad a \notin \text{dom } S \quad \forall j < |O_s|, i \neq j \rightarrow a \notin \text{dom } R_{s[j]}
\]

\[
O_s, R_s, i \vdash (\text{WRITE } \text{False } a \ (D, f) \ A \ L \ R \ W \ \# \ is, \ \emptyset, m, D, O, S) \checkmark
\]

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

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

\[
O_s, R_s, i \vdash (\text{WRITE } \text{True } a \ (D, f) \ A \ L \ R \ W \ \# \ is, \ \emptyset, m, D, O, S) \checkmark
\]

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

\[
O_s, R_s, i \vdash (\text{RMW } a \ (D, f) \ \text{cond } \text{ret } A \ L \ R \ W \ \# \ is, \ \emptyset, m, D, O, S) \checkmark
\]

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

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

\[
O_s, R_s, i \vdash (\text{FENCE } \# \ is, \ \emptyset, m, D, O, S) \checkmark
\]

\[
O_s, R_s, i \vdash (\text{GHOST } A \ L \ R \ W \ \# \ is, \ \emptyset, m, D, O, S) \checkmark
\]

\[
O_s, R_s, i \vdash ([], \ \emptyset, m, D, O, S) \checkmark
\]

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

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

\[
O' = O \cup A - R \quad S' = S \oplus_w R \oplus_A L
\]

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

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

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

\[
O' = O \cup A - R \quad R' = \text{aug } (\text{dom } S) R \quad S' = S \oplus_w R \oplus_A L
\]

\[
(m, \text{GHOST}_{ab} A \ L \ R \ W \ \# \ sb, O, R, S) \rightarrow_{ab} (m, \text{sb}, O', R', S')
\]

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 \text{\(sb' = sb \odot [\text{READ}_{sb} \text{volatile \(a\) \(t\)} \(v\)]\)}
\]

\[
\text{(Read volatile \(a\) \# is, \(\vartheta\), \(sb\), m, \(ghst\))} \xrightarrow{abh_m} (\text{is}, \(\vartheta(t \mapsto v)\), \(sb'\), m, \(ghst\))
\]

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

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

\[
\text{(RMW \(a\) \(t\) \((D, f)\) \(\text{cond ret} A\) \(L\) \(R\) \(W\) \# is, \(\vartheta\), \([\cdot]\), m, \(ghst\))} \xrightarrow{abh_m} (\text{is}, \(\vartheta(t \mapsto m\ a)\) \, \((f\ (\vartheta(t \mapsto m\ a)))\) \quad \text{\(m' = m(a := f\ (\vartheta(t \mapsto m\ a)))\)}
\]

\[
\text{\(ghst = (D, O, R, S)\)} \quad \text{\(ghst' = (\text{false}, O, A - R, \text{empty}, S \odot_{\text{w}} R \odot_{\text{A}} L)\)}
\]

\[
\text{(Fence \# is, \(\vartheta\), \([\cdot]\), m, \(D, O, R, S\) \xrightarrow{abh_m} (\text{is}, \(\vartheta\), \([\cdot]\), m, \text{false}, O, \text{empty}, S)}
\]

\[
\text{(Ghost \(A\) \(L\) \(R\) \# is, \(\vartheta\), \(sb\), m, G) \xrightarrow{abh_m} (\text{is}, \(\vartheta\), \(sb\) \odot [\text{Ghost}_{sb} \(A\) \(L\) \(R\) \(W\), m, G])}
\]

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

are analogous to the rules in Figure 2 replacing the memory transions and store buffer transistions 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 threads 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.

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

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:

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

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 \texttt{execs} and the suspended ones by \texttt{suspends}. The function \texttt{instrs} converts them back to instructions, which are a prefix of the instructions of the virtual
reads from θ hd-prog program state recorded in the suspended part of the store buffer. This state is selected by the state of the virtual machine is either the same as in the store buffer machine or the first suspicious is @ but not yet on the virtual machine. This situation is formalized as

\[ \text{suspends} = \text{dropWhile not-volatile-write sb} \]

\[ \text{instructions have already made their way to the instructions of the store buffer machine. We collect the additional instructions which were issued by program instructions exec-all-until-volatile-write} \]

\[ \text{local, and hence could be executed in any order with the same result on the memory. Function exec-all-until-volatile-write executes them in order of appearance. Similarly the sharing map of the virtual machine is obtained by executing all store buffers until the first volatile write via the function share-all-until-volatile-write. For the local ownership set } \]

\[ \text{O}_{\text{sbh}} \text{ the auxiliary function acquire calculates the outstanding effect of the already simulated parts of the store buffer. Analogously release calculates the effect for the released addresses } \]

\[ \text{R}_{\text{sbh}} \].

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).**

\[ (ts_{abh}, m_{abh}, S_{abh}) \stackrel{sbh}{\rightarrow} (ts_{abh}', m_{abh}', S_{abh}') \land (ts_{abh}, m_{abh}, S_{abh}) \sim (ts, m, S) \land \]

\[ \text{safe-reach-delayed (ts, m, S) } \land \text{invariant ts}_{abh} \text{ S}_{abh} \text{ m}_{abh} \rightarrow \]

\[ \text{invariant ts}_{abh}' \text{ S}_{abh}' \text{ m}_{abh}' \land \]

\[ (\exists ts' S' m') \sim (ts, m, S) \rightarrow (ts', m', S') \land (ts_{abh}', m_{abh}', S_{abh}') \sim (ts', m', S') \]

In the following we discuss the invariant \(ts_{sbh} S_{sbh} m_{sbh}\), where we commonly refer to a thread configuration \(ts_{sbh} [i] = (p, is, θ, sb, D, O, R)\) for \(i < |ts_{sbh}|\). 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:

\[ \text{invariant ts}_{sbh} S_{sbh} m_{sbh} \equiv \text{ownership-inv S}_{sbh} \text{ ts}_{sbh} \land \text{sharing-inv S}_{sbh} \text{ ts}_{sbh} \land \]

\[ \]
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 valid-sop \((D, f)\), i.e., function \(f\) only depends on domain \(D\). (ii) For every suffix of the instructions of the form \texttt{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 \texttt{Write} \(a\) \( (D, f) v A L R W\#\) 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 \texttt{PROGab} \(p_1 p_2 \# sb'\), 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 \(\vartheta | (\neg \text{read-tmps} sb') \vdash p_1 \rightarrow_p (p_2, is')\) 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: \(\exists is'. \text{instrs} sb' @ is = is' @ \text{prog-instrs} sb'\), i.e., the remaining instructions \(is\) correspond to a suffix of the recorded instructions \(\text{prog-instrs} sb'\).

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 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 valid. Program steps, memory steps and store buffer steps have to maintain 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 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 exists 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 $c_{sbh} = (ts_{sbh}, m_{sbh}, S_{sbh})$ of the store buffer machine, where the next instruction to be executed is a read of thread $i$ \texttt{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 $(ts_{sbh}, m_{sbh}, S_{sbh}) \sim (ts, m, S)$, the invariants $t_{sbh} \ S_{sbh} \ m_{sbh}$ and the safety of all reachable states of the virtual machine: 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 \(cfg'' = cfg\). We have to show that the virtual machine reads the same value from memory: \(v = m a\). So what can go wrong? When can 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.

![Fig. 16: Conflicting write in thread \(j\) (marked \(x\))]({})

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 \((\text{Ownership}.i)\) and \((\text{Sharing}.i)\). Additionally from invariant \((\text{Sharing}.v)\) 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 \(cfg\): 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 \(a\) 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 (neither 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 \(cfg''\) 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 \( \text{cfg}'' \). However, consider an ghost operation that acquires an address \( a \). From safety of the configuration \( \text{cfg}'' \) 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 \( \text{cfg}'' \) 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 \text{exec-all-until-volatile-write} \ and \text{share-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 has 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 an 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 \((ts_{sb}, m) \xrightarrow{sb}^* (ts_{sb}'', m')\) by a computation of a store buffer machine with history \((ts_{sbh}, m, S) \xrightarrow{sbh}^* (ts_{sbh}'', m'', S'')\),
2. simulate the computation of the store buffer machine with history by a computation of the virtual machine with delayed releases \((ts, m, S) \xrightarrow{vb}^* (ts', m', S')\) 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 \((ts, m, S) \xrightarrow{v}^* (ts', m', S')^5\).

Footnote 5: Here we are sloppy with \( ts \); strictly we would have to distinguish the thread configurations without the \( R \) component form 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 $\mathcal{R}$ ghost component can be ignored in Step 3. However, to apply Theorem 2 in Step 2 we have to convert from $\text{safe-reach} (ts, m, S)$ provided by the preconditions of Theorem 1 to the required $\text{safe-reach-delayed} (ts, m, S)$. 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 $\mathcal{R}$ 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 simulataneously 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) $\text{Const} \, v$, a constant value, (ii) $\text{Mem} \, \text{volatile} \, a$, a (volatile) memory lookup at address $a$, (iii) $\text{Tmp} \, \text{sop}$, reading from the temporaries with a operation $\text{sop}$ 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 expression \( a \), (iii) **CAS** \( a \ c_ e \ s_e \) \( A L R W \), atomic compare and swap at address expression \( a \) with compare expression \( c_ e \) and swap expression \( s_e \), (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 \( \text{used-tmps} \ e \) calculates the number of temporaries that are necessary to evaluate expression \( e \), where every **MEM** expression accounts to one temporary. With **issue-expr** \( t \ e \) we obtain the instruction list for expression \( e \) starting at temporary \( t \), whereas **eval-expr** \( t \ e \) 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 condition \( e \) 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 \( s_e \) 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
∀ sop. a \neq \text{TMP sop } \quad a' = \text{TMP (eval-expr } t \ a) \quad t' = t + \text{used-tmps } a \quad \text{is = issue-expr } t \ a

\vdash (\text{ASSIGN \ volatile } a \ e \ \text{A L R W}, \ t) \rightarrow p ((\text{ASSIGN \ volatile } a' \ e \ \text{A L R W}, \ t'), \ \text{is})

D \subseteq \text{dom } \varnothing \quad \text{is = issue-expr } t \ e @ [\text{WRITE \ volatile } (a \ \varnothing) \ (\text{eval-expr } e \ t \ (A \ \varnothing) \ (L \ \varnothing) \ (R \ \varnothing) \ (W \ \varnothing)]

\vdash (\text{ASSIGN \ volatile } (\text{TMP } (D, \ a)) \ e \ \text{A L R W}, \ t) \rightarrow p ((\text{Skip}, \ t + \text{used-tmps } e), \ \text{is})

∀ sop. a \neq \text{TMP sop } \quad a' = \text{TMP (eval-expr } t \ a) \quad t' = t + \text{used-tmps } a \quad \text{is = issue-expr } t \ a

\vdash (\text{CAS } a \ c_a \ s_a \ \text{A L R W}, \ t) \rightarrow p ((\text{CAS } a' \ c_a' \ s_a \ \text{A L R W}, \ t'), \ \text{is})

\vdash (\text{TMP } (\text{D_k} \ a)) (\text{TMP } (D_e, \ c)) s_e \ \text{A L R W}, \ t) \rightarrow p ((\text{Skip}, \ Suc \ t'), \ \text{is})

\text{D_k} \subseteq \text{dom } \varnothing \quad \text{eval-expr } s_e = (D, \ t) \quad t' = t + \text{used-tmps } s_e \quad \text{cond} = (\lambda \varnothing. \text{the } (\varnothing \ t') = c \ \varnothing)

\text{ret} = (\lambda v_1, v_2, v_1) \quad \text{is = issue-expr } t \ s_e @ [\text{RMW } (a \ \varnothing) \ t' (D, \ t) \ \text{cond } \text{ret} (A \ \varnothing) (L \ \varnothing) (R \ \varnothing) (W \ \varnothing)]

\vdash (\text{CAS } (\text{TMP } (D_k, \ a)) \ (\text{TMP } (D_e, \ c)) s_e \ \text{A L R W}, \ t) \rightarrow p ((\text{Skip}, \ Suc \ t'), \ \text{is})

\text{D_e} \subseteq \text{dom } \varnothing \quad \text{SEQ } s_1 \ s_2, \ t \rightarrow p ((\text{SEQ } s_1 \ s_2, \ t'), \ \text{is})

\text{SEQ } \text{SKIP } s_2, \ t \rightarrow p ((s_2, \ t), [])

∀ sop. e \neq \text{TMP sop } \quad e' = \text{TMP (eval-expr } t \ e) \quad t' = t + \text{used-tmps } e \quad \text{is = issue-expr } t \ e

\vdash (\text{COND } e \ s_1 \ s_2, \ t) \rightarrow p ((\text{COND } e' \ s_1 \ s_2, \ t'), \ \text{is})

\text{D} \subseteq \text{dom } \varnothing \quad \text{isTrue } (e \ \varnothing)

\text{D} \subseteq \text{dom } \varnothing \quad \neg \text{isTrue } (e \ \varnothing)

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

\vdash (\text{SGHOST } \text{A L R W}, \ t) \rightarrow p ((\text{Skip}, \ t), [\text{Ghost } (A \ \varnothing), (L \ \varnothing), (R \ \varnothing), (W \ \varnothing)])

\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 \( \vartheta \) be the valuation of temporaries in the current configuration, for every thread configuration \( s_{sb}[i] = ((s, t), i, \vartheta, sb, D, O) \) where \( i < |s_{sb}| \) we require: (i) The domain of all intermediate 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 \( \vartheta \) 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
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.θ.
  [sop=(D,f); D ⊆ dom θ]⇒
  f θ = f (θ|'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
**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

**type-synonym** 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
locale program =
  fixes
  program-step :: tmprs ⇒ 'p ⇒ 'p × instrs ⇒ bool (-⇒ -σ - [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 × tmprs × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒
  (instrs × tmprs × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒ bool and
program-step :: tmprs ⇒ '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
  (- ⇒ -σ - [60,60] 100)
where

Program:
[i < length ts; ts!i = (p,is,σ,sb,D,O,R);
  σ+p ⇒ p (p′,is′)] ⇒⇒
(ts,m,S) ⇒ (ts[i]=(p′,is′(is),σ,record p p′ is′ sb,D,O,R)|m,S)

| Memop:
[i < length ts; ts!i = (p,is,σ,sb,D,O,R);
  (is,σ,σ,σ,m,D,O,R,S) m⇒ (is′,σ,σ,σ,m′,D′,O′,R′,S′)]
⇒⇒
(ts,m,S) ⇒ (ts[i]=(p,is′,σ,σ,σ,m′,D′,O′,R′)|m′,S′)

| StoreBuffer:
[i < length ts; ts!i = (p,is,σ,σ,σ,m,D,O,R);
  (m,σ,σ,σ,m,D,O,R,S) m⇒ (m′,σ,σ,σ,m′,D′,O′,R′,S′)]
⇒⇒
(ts,m,S) ⇒ (ts[i]=(p,is,σ,σ,σ,m′,D′,O′,R′)|m′,S′)

definition final:: ( 'p,'sb,'dirty,'owns,'rels,'shared) global-config ⇒ bool
where
final c = (¬ (∃ c′. c ⇒ c′))
lemma store-buffer-steps:
assumes sb-step: storebuffer-step\(\ast\ast\) \((m, sb, O, R, S)\) \((m', sb', O', R', S')\)
shows \(\forall ts. i < \text{length } ts \implies ts!i = (p, is, \emptyset, sb, D, O, R) \implies\)
\(\text{concurrent-step}\(\ast\ast\) \((ts, m, S)\) \((ts[i := (p, is, \emptyset, sb', D, O', R')], m', S')\)
using sb-step
proof (induct rule: converse-rtranclp-induct5)
case refl then show \(?\)case
  by (simp add: list-update-id')
next
case (step m sb ORS m' sb' O' R' S')
  note i-bound = \(i < \text{length } ts\)
  note step = \((m, sb, O, R, S) \rightarrow sb\) \((m', sb', O', R', S')\)
  let \(?ts' = ts[i := (p, is, \emptyset, sb', D, O', R')]\)
  from StoreBuffer \((OF i-bound ts-i step)\)
  have \((ts, m, S) \implies (?ts', m', S')\).
  also
  from i-bound have i-bound': \(i < \text{length } ?ts'\) by simp
  from i-bound have ts'-i: \(?ts'!i = (p, is, \emptyset, sb', D, O', R')\)
    by simp
  from step.hyps (3) \((OF i-bound' ts'-i) i-bound\)
  have concurrent-step\(\ast\ast\) \((?ts', m'', S'')\) \((ts[i := (p, is, \emptyset, sb', D, O', R')], m', S')\)
    by (simp)
  finally
  show \(?\)case .
qed

lemma step-preserves-length-ts:
assumes step: \((ts, m, S) \implies (ts', m', S')\)
shows \(\text{length } ts = \text{length } ts'\)
using step
proof (cases)
apply \(\text{cases}\)
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 \((- \oplus - [61,1000,60] 61)\)
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
\(S \ominus_A L \equiv (\lambda a. \text{if } a \in L \text{ then None else } (\text{case } S a \text{ of } \text{None } \Rightarrow \text{None}))\)
definition read-only :: shared ⇒ addr set
where
read-only S ≡ {a. (S a = Some False)}

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

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

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

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

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

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

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

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

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

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

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

lemma restrict-shared-out: a ∉ L ⟹ (m ⊖ₚ L) a = map-option (λwriteable. (a ∈ A ∨ writeable)) (m a)
  by (auto simp add: restrict-shared-def split: if-split-asn option.splits)

lemma restrict-shared-out'[simp]:
  a ∉ L ⟹ m a = Some writeable ⟹ (m ⊖ₚ L) a = Some (a ∈ A ∨ writeable)
  by (simp add: restrict-shared-out)
lemma augment-mono-map: \( A \subseteq_m B \Rightarrow (A \oplus C) \subseteq_m (B \oplus C) \)
by (auto simp add: augment-shared-def map-le-def domIff)

lemma augment-mono-map: \( A \subseteq_s B \Rightarrow (A \oplus C) \subseteq_s (B \oplus C) \)
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: \( A \subseteq_s B \Rightarrow (A \ominus C) \subseteq_s (B \ominus C) \)
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: \( \text{dom } A \subseteq \text{dom } B \Rightarrow \text{dom } (A \oplus C) \subseteq \text{dom } (B \oplus C) \)
by auto

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

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

lemma in-read-only-restrict-conv:
\[
a \in \text{read-only } (S \ominus_A L) = (a \in \text{read-only } S \wedge a \notin L \wedge a \notin A)
\]
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)

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:: addr set \Rightarrow addr set \Rightarrow rels \Rightarrow rels
where
augment-rels \( S \rightarrow R \) = (\( \lambda a \). if \( a \in R \) then (case \( R \) a of None ⇒ Some (a ∈ S) | Some s ⇒ Some (s ∧ (a ∈ S))) else \( R \) a)

declare domIf [iff del]

A.3 Memory Transitions

locale gen-direct-memop-step = fixes emp::\( \mathbf{'} \)rels and aug::owns ⇒ rel ⇒ \( \mathbf{'} \)rels ⇒ \( \mathbf{'} \)rels begin inductive gen-direct-memop-step :: (instrs × tmps × unit × memory × bool × owns × \( \mathbf{'} \)rels × shared ) ⇒ (instrs × tmps × unit × memory × bool × owns × \( \mathbf{'} \)rels × shared ) ⇒ bool (- → - [60,60] 100) where Read: (Read volatile a t # is \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \) (t→m a), x, m, \( D \), \( O \), \( R \), \( S \))

| WriteNonVolatile: (Write False a (D,f) A L R W#is, \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \), x, m(a := f \( \theta \)), \( D \), \( O \), \( R \), \( S \))

| WriteVolatile: (Write True a (D,f) A L R W# is \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \), x, m(a:=f \( \theta \)), True, \( O \) ∪ A − R, emp, \( S \) ⊕ \( W \) \( R \) ⊖ \( A \) \( L \))

| Fence: (Fence # is \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \), x, m, False, \( O \), emp, \( S \))

| RMWReadOnly: [¬ cond (\( \theta \)(t→m a))] \( \implies \) (RMW a t (D,f) A L R W# is \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \)(t→m a),x,m, False, \( O \), emp, \( S \))

| RMWWrite: [cond (\( \theta \)(t→m a))] \( \implies \) (RMW a t (D,f) cond ret A L R W# is \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \)(t→ret (m a) (f(\( \theta \)(t→m a)))),x, m(a:= f(\( \theta \)(t→m a)))), False,\( O \) ∪ A − R, emp, \( S \) ⊕ \( W \) \( R \) ⊖ \( A \) \( L \))

| Ghost: (Ghost A L R W# is \( \theta \), x, m, \( D \), \( O \), \( R \), \( S \)) → (is, \( \theta \), x, m, \( D \), \( O \) ∪ A − R, aug (dom \( S \) R \( R \) , \( S \) ⊕ \( W \) \( R \) ⊖ \( A \) \( L \)) end interpretation direct-memop-step: gen-direct-memop-step Map.empty augment-rels .
**term** direct-memop-step.gen-direct-memop-step

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

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

(instrs × tmps × unit × memory × bool × owns × rels × shared )

⇒ bool

(- →* - [60,60] 100)

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

term x →^* Y

**interpretation** virtual-memop-step: gen-direct-memop-step () (λS R. ()).

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

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

(- →* - [60,60] 100)

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

term x →_v Y

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

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

(instrs × tmps × 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, θ, x, m, D, O, R, S) → (is', θ', x', m', D', O', R', S')

**shows** (is, θ, x, m, D, O, (), S) →_v (is', θ', 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

**inductive** safe-direct-memop-state :: owns list ⇒ nat ⇒
(instrs × tmpls × memory × bool × owns × shared) ⇒ bool
(=-,-⊢ - √ \[60,60,60\] 100)

**where**
Read: \[ a ∈ O ∨ a ∈ read-only S ∨ (volatile ∧ a ∈ dom S); \]
volatile → ¬ D]
⇒ Os,i⊢(Read volatile a t # is, ∅, m, D, O, S)√

| WriteNonVolatile: [a ∈ O; a /∈ dom S]
⇒ Os,i⊢(Write False a (D,f) A L R W#is, ∅, m, D, O, S)√

| WriteVolatile:
[∀ j < length Os. i\# j → a /∈ Os\#j; A ⊆ dom S ∪ O; L \subseteq A; R \subseteq O; A ∩ R = {};
∀ j < length Os. i\# j → A ∩ Os\#j = {};
a /∈ read-only S]
⇒ Os,i⊢(Write True a (D,f) A L R W# is, ∅, m, D, O, S)√

| Fence:
Os,i⊢(Fence # is, ∅, m, D, O, S)√

| Ghost:
A \subseteq dom S ∪ O; L \subseteq A; R \subseteq O; A ∩ R = {};
⇒ Os,i⊢(Ghost A L R W# is, ∅, m, D, O, S)√

| RMWReadOnly:
[- cond (\(θ(t→m a))\); a ∈ O ∨ a ∈ dom S] \implies
Os,i⊢(RMW a t (D,f) cond ret A L R W# is, ∅, m, D, O, S)√

| RMWWrite:
[cond (\(θ(t→m a))\);
∀ j < length Os. i\# j → a /∈ Os\#j;
A \subseteq dom S ∪ O; L \subseteq A; R \subseteq O; A ∩ R = {};
∀ j < length Os. i\# j → A ∩ Os\#j = {};
a /∈ read-only S]
⇒ Os,i⊢(RMW a t (D,f) cond ret A L R W# is, ∅, m, D, O, S)√

| Nil: Os,i⊢([], ∅, m, D, O, S)√
**inductive** safe-delayed-direct-memop-state :: owns list ⇒ rels list ⇒ nat ⇒
(intrs × tmps × memory × bool × owns × shared) ⇒ bool

(\cdot,\cdot,\cdot) - √ [60,60,60,60] 100

**where**

Read: \[a ∈ O \lor a ∈ \text{read-only } S \lor (\text{volatile } \land a ∈ \text{dom } S)\];
\[∀ j < \text{length } Os. i \neq j \rightarrow (Rs[j]) a \neq \text{Some False} \land \text{ no release of unshared address}\]
\[\text{volatile} \rightarrow (\forall j < \text{length } Os. i \neq j \rightarrow a \notin \text{dom } (Rs[j]))\];
\[\text{volatile} \rightarrow \neg D \]
\[⇒ Os,Rs,i ⊩ (\text{Read } a) t # \text{ is, } \emptyset, m, D, O, S)√\]

WriteNonVolatile:
\[a ∈ O; a \notin \text{dom } S; ∀ j < \text{length } Os. i \neq j \rightarrow a \notin \text{dom } (Rs[j])\]
\[⇒ Os,Rs,i ⊩ (\text{Write False } a) A L R W#is, \emptyset, m, D, O, S)√\]

WriteVolatile:
\[∀ j < \text{length } Os. i \neq j \rightarrow a \notin (Os[j] \cup \text{dom } (Rs[j]))\];
\[A ⊆ \text{dom } S \cup O; L ⊆ A; R ⊆ O; A \cap R = \{\}\];
\[∀ j < \text{length } Os. i \neq j \rightarrow A \cap (Os[j] \cup \text{dom } (Rs[j])) = \{\}\];
\[a \notin \text{read-only } S]\]
\[⇒ Os,Rs,i ⊩ (\text{Write True } a) A L R W# is, \emptyset, m, D, O, S)√\]

Fence:
\[Os,Rs,i ⊩ (\text{Fence } #) is, \emptyset, m, D, O, S)√\]

Ghost:
\[A ⊆ \text{dom } S \cup O; L ⊆ A; R ⊆ O; A \cap R = \{\}\];
\[∀ j < \text{length } Os. i \neq j \rightarrow A \cap (Os[j] \cup \text{dom } (Rs[j])) = \{\}\]
\[⇒ Os,Rs,i ⊩ (\text{Ghost } A L R W# is, \emptyset, m, D, O, S)√\]

RMWReadOnly:
\[\neg \text{cond (}(\emptyset (t \rightarrow m) a)) ; a ∈ O \lor a ∈ \text{dom } S\];
\[∀ j < \text{length } Os. i \neq j \rightarrow (Rs[j]) a \neq \text{Some False} \land \text{ no release of unshared address}\]
\[⇒ Os,Rs,i ⊩ (\text{RMW } a) t (D,f) \text{ cond ret } A L R W# is, \emptyset, m, D, O, S)√\]

RMWWrite:
\[\text{cond (}(\emptyset (t \rightarrow m) a)) ; a ∈ O \lor a ∈ \text{dom } S\];
\[∀ j < \text{length } Os. i \neq j \rightarrow a \notin (Os[j] \cup \text{dom } (Rs[j]))\];
\[A ⊆ \text{dom } S \cup O; L ⊆ A; R ⊆ O; A \cap R = \{\}\];
\[∀ j < \text{length } Os. i \neq j \rightarrow A \cap (Os[j] \cup \text{dom } (Rs[j])) = \{\}\];
\[a \notin \text{read-only } S]\]
\[⇒ Os,Rs,i ⊩ (\text{RMW } a) t (D,f) \text{ cond ret } A L R W# is, \emptyset, m, D, O, S)√\]

Nil:
\[Os,Rs,i ⊩ ([], \emptyset, m, D, O, S)√\]
**Lemma** memop-safe-delayed-implies-safe-free-flowing:
assumes safe-delayed: $\mathcal{O}_s, \mathcal{R}_s, i \vdash (\text{is}, \theta, m, D, \mathcal{O}, \mathcal{S})\checkmark$
shows $\mathcal{O}_s, i \vdash (\text{is}, \theta, m, D, \mathcal{O}, \mathcal{S})\checkmark$
using safe-delayed

**Proof** (cases)
- **Case** Read **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.intros)

- **Case** WriteNonVolatile **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.intros)

- **Case** WriteVolatile **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.intros)

- **Case** Fence **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.intros)

- **Case** Ghost **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.Ghost)

- **Case** RMWReadOnly **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.RMWWrite)

- **Case** RMWWrite **thus** ?thesis
  by (fastforce intro!: safe-direct-memop-state.RMWWrite)

- **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: $\mathcal{O}_s, i \vdash (\text{is}, \theta, m, D, \mathcal{O}, \mathcal{S})\checkmark$
assumes empty: $\forall R \in \text{set } \mathcal{R}_s. R = \text{Map}.\text{empty}$
assumes leq: length $\mathcal{O}_s = \text{length } \mathcal{R}_s$
assumes unowned-shared: $\forall a. (\forall i < \text{length } \mathcal{O}_s. a \notin (\mathcal{O}_s!i)) \longrightarrow a \in \text{dom } \mathcal{S}$
asumes $\mathcal{O}_s!i = \mathcal{O}$
shows $\mathcal{O}_s, \mathcal{R}_s, i \vdash (\text{is}, \theta, m, D, \mathcal{O}, \mathcal{S})\checkmark$
using safe

**Proof** (cases)
- **Case** Read **thus** ?thesis
  using leq empty
  by (fastforce intro!: safe-delayed-direct-memop-state.Read dest: nth-mem)

- **Case** WriteNonVolatile **thus** ?thesis
  using leq empty
  by (fastforce intro!: safe-delayed-direct-memop-state.intros dest: nth-mem)

- **Case** WriteVolatile **thus** ?thesis
  using leq empty
  apply clarsimp

41
apply (rule safe-delayed-direct-memop-state.WriteVolatile)
apply (auto)
apply (drule nth-mem)
apply fastforce
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
  by (fastforce intro!: safe-delayed-direct-memop-state.intros dest: nth-mem)
next
case (RMWWrite cond t a A L R D f ret W) 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 (- → (60,60] → 100)
where
Id: (m,x,O,R,S) → (m,x,O,R,S)
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
(- ⇒ₕ [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
(- ⇒ₕ* [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
(- ⇒ₐ [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
(- ⇒ₐ* [60,60] 100)
where
virtual-concurrent-steps == virtual-concurrent-step^*^*

term x ⇒ₐ Y
term x ⇒ₐ* Y
term x ⇒ₐ* Y

definition safe-reach step safe cfg ≡
  ∀ cfg', step^*^* cfg cfg' −→ safe cfg'
lemma safe-reach-safe-refl: safe-reach step safe cfg \implies safe cfg
apply (auto simp add: safe-reach-def)
done

lemma safe-reach-safe-rtrancl: safe-reach step safe cfg \implies step^{**} cfg cfg' \implies safe cfg'
  by (simp only: safe-reach-def)

lemma safe-reach-steps: safe-reach step safe cfg \implies step^{**} cfg cfg' \implies safe-reach step safe cfg'
apply (auto simp add: safe-reach-def intro: rtranclp-trans)
done

lemma safe-reach-step: safe-reach step safe cfg \implies step cfg cfg' \implies safe-reach step safe cfg'
apply (erule safe-reach-steps)
apply (erule r-into-rtranclp)
done

color context program
begin

abbreviation safe-reach-direct \equiv safe-reach direct-concurrent-step

lemma safe-reac-direct-def: safe-reach-direct safe cfg \equiv 
\forall cfg'. cfg \Rightarrow d^* cfg' \rightarrow safe cfg'
by (simp add: safe-reach-def)

abbreviation safe-reach-virtual \equiv safe-reach virtual-concurrent-step

lemma safe-reac-virtual-def: safe-reach-virtual safe cfg \equiv 
\forall cfg'. cfg \Rightarrow v^* cfg' \rightarrow safe cfg'
by (simp add: safe-reach-def)
end

definition safe-free-flowing cfg \equiv let (ts,m,S) = cfg 
in (\forall i < \text{length ts}. \text{let } (p,is,\emptyset,x,D,O,R) = ts!i \text{ in } 
\text{map owned ts,i } \vdash (is,\emptyset,m,D,O,S)\sqrt{)}

lemma safeE: [safe-free-flowing (ts,m,S);i<\text{length ts}; ts!i=(p,is,\emptyset,x,D,O,R)] 
\implies \text{map owned ts,i } \vdash (is,\emptyset,m,D,O,S)\sqrt{
by (auto simp add: safe-free-flowing-def)
safe-delayed $cfg \equiv \text{let } (ts, m, S) = cfg$

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

\[ \text{map owned } ts, \text{map released } ts, i \vdash (is, \emptyset, m, D, O, S) \vee \]

**lemma** safe-delayedE: $[\text{safe-delayed } (ts, m, S); i < \text{length } ts; ts!i = (p, is, \emptyset, x, D, O, R)]$

\[ \implies \text{map owned } ts, \text{map released } ts, i \vdash (is, \emptyset, m, D, O, S) \vee \]

by (auto simp add: safe-delayed-def)

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

**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**

\[ \text{interpret } \text{direct-computation:} \]

\[ \text{interpret } \text{direct-memop-step empty-storebuffer-step program-step } \lambda p p' \text{ is sb. sb.} \]

\[ \text{interpret } \text{virtual-computation:} \]

\[ \text{interpret } \text{virtual-memop-step empty-storebuffer-step program-step } \lambda p p' \text{ is sb. sb.} \]

**from** step

**show** $?\text{thesis}$

**proof** (cases)

**case** $(\text{Program } j \ p \ is \ \emptyset \ sb \ D \ O \ R \ p' \ is')$

**then obtain**

\[ ts': ts' = ts[j := (p', is' @ is', \emptyset, sb, D, O, R)] \text{ and } \]

\[ S': S = S \text{ and } \]

\[ m': m' = m \text{ and } \]

\[ j \text{-bound: } j < \text{length } ts \text{ and } \]

\[ ts-j: ts!j = (p, is, \emptyset, sb, D, O, R) \text{ and } \]

\[ \text{prog-step: } \emptyset \vdash p \rightarrow_p (p', is') \]

by auto

**from** ts-j j-bound **have**

\[ \text{vts-j: remove-rels } ts!j = (p, is, \emptyset, sb, D, O, ()) \text{ by (auto simp add: remove-rels-def)} \]

**from** virtual-computation.$\text{Program } [\text{OF } - \text{vts-j prog-step, of m S}] \text{ j-bound } ts'$

**show** $?\text{thesis}$

by (clarsimp simp add: $S' \text{ m' } remove-rels-def$ map-update)

**next**

**case** $(\text{Memop } j \ p \ is \ \emptyset \ sb \ D \ O \ R \ is' \ \emptyset' \ sb' \ D' \ O' \ R')$

**then obtain**

\[ ts': ts' = ts[j := (p, is', \emptyset', sb', D', O', R')] \text{ and } \]

\[ j \text{-bound: } j < \text{length } ts \text{ and } \]

\[ ts-j: ts!j = (p, is, \emptyset, sb, D, O, R) \text{ and } \]

\[ \text{mem-step: } (is, \emptyset, sb, m, D, O, R, S) \rightarrow (is', \emptyset', sb', m', D', O', R', S') \]

by auto

**from** ts-j j-bound **have**

\[ \text{vts-j: remove-rels } ts!j = (p, is, \emptyset, sb, D, O, ()) \text{ by (auto simp add: remove-rels-def)} \]

**from** virtual-computation.$\text{Memop } [\text{OF } - \text{vts-j prog-step, of m S}] \text{ j-bound } ts'$

virtual-memop-step-simulates-direct-memop-step $[\text{OF mem-step}] \text{ j-bound } ts'$
show thesis
   by (clarsimp simp add: remove-rels-def map-update)
next
  case (StoreBuffer - p is \emptyset \ D \ O \ \mathcal{R} \ sb' \ O' \ \mathcal{R}')
hence False
   by (auto simp add: empty-storebuffer-step-def)
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 (remove-rels ts',m',S')
using steps
proof (induct rule: converse-rtranclp-induct-sbh-steps)
  case refl
   thus thesis 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 (remove-rels ts',m',S')
    by blast
  note virtual-simulates-direct-step [OF first]
  also note hyp
  finally
   show thesis by blast
qed

locale simple-ownership-distinct =
  fixes ts::('p,\mathcal{D},\mathcal{O},\mathcal{R},\mathcal{S}) thread-config list
  assumes simple-ownership-distinct:
    \[ \forall i,j \in \{0\ldots\text{length ts}\} \, i \neq j \rightarrow \text{let } (p_i,\text{is}_i,\text{sb}_i,\text{D}_i,\text{O}_i,\text{R}_i) = ts!i \text{ in } \text{let } (p_j,\text{is}_j,\text{sb}_j,\text{D}_j,\text{O}_j,\text{R}_j) = ts!j \text{ in } (\text{O}_i \cap \text{O}_j) = \{\} \] 

lemma (in simple-ownership-distinct)
  simple-ownership-distinct-nth-update:
  \[ \forall i \in \{0\ldots\text{length ts}\} \rightarrow \text{let } (p_i,\text{is}_i,\text{sb}_i,\text{D}_i,\text{O}_i,\text{R}_i) = ts!i \text{ in } (\text{O}_i \cap \text{O}_j) = \{\} \] 

apply (unfold-locales)
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)
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:
\[ \bigwedge i \ p \ \text{is} \ \text{ORD} \ \theta \ \text{sb}. \ [i < \text{length ts}; ts!i = (p,\text{is},\theta,\text{sb},D,O,R) ] \implies \bigcap O \cap \text{read-only} \ S = \{\}\]

lemma (in read-only-unowned)
read-only-unowned-nth-update:
\[ \bigwedge i \ p \ \text{is} \ \text{ORD} \ \text{acq} \ \theta \ \text{sb}. \ [i < \text{length ts}; O \cap \text{read-only} \ S = \{\}] \implies \text{read-only-unowned} (S(ts[i := (p,\text{is},\theta,\text{sb},D,O,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:
\[ \neg \bigcup ((\lambda (-,-,-,-,-O,-). O) \ \text{set ts}) \subseteq \text{dom} S \]

lemma (in unowned-shared)
unowned-shared-nth-update:
\[ \bigwedge i \ p \ \text{is} \ \text{ORD} \ \text{acq} \ \theta \ \text{sb}. \ [i < \text{length ts}; \text{O} \subseteq \text{O'}] \implies \text{unowned-shared} (S(ts[i := (p',\text{is}',\text{xs}',\text{sb}',\text{D}',O',R')]])
proof
- from i-bound ith subset
  have \[ \bigcup ((\lambda (-,-,-,-,-O,-). O) \ \text{set ts}) \subseteq \bigcup ((\lambda (-,-,-,-,-O,-). O) \ \text{set (ts[i := (p',\text{is}',\text{xs}',\text{sb}',\text{D}',O',R')]])}\]
  apply (auto simp add: in-set-conv-nth nth-list-update split: if-split-asm)
subgoal for x p'' is'' xs'' sb'' D'' O'' R'' j
apply (case-tac j=i)
apply (rule-tac x=(p',\text{is}',\text{xs}',\text{sb}',\text{D}',O',R')] in bexI)
apply fastforce
apply (fastforce simp add: in-set-conv-nth)
apply (rule-tac x=(p'',is'',xs'',sb'',D'',O'',R''] in bexI)
apply fastforce
apply (fastforce simp add: in-set-conv-nth)
done
done
hence \[ \neg \bigcup ((\lambda (-,-,-,-,-O,-). O) \ \text{set (ts[i := (p',\text{is}',\text{xs}',\text{sb}',\text{D}',O',R')]}) \subseteq \neg \bigcup ((\lambda (-,-,-,-,-O,-). O) \ \text{set ts}) \]
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 < length ts
  assumes ts-i: ts!i = (p, is, 0, sb, D, O, R)
  assumes a-unowned-others:
    \forall j < length (map owned ts). i \neq j \rightarrow
    (let O_j = (map owned ts)!j in a \notin O_j)

  shows a \in O \lor a \in dom S
proof -
{  
  fix j p_j is_j R_j D_j xs_j sb_j
  assume a-unowned: a \notin O
  assume j-bound: j < length ts
  assume jth: ts!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
  have a \notin O_j
    proof (cases i=j)
      case True with a-unowned-i jth
        show ?thesis
          by auto
    next
      case False from a-unowned-others [rule-format, of j] j-bound jth False
        show ?thesis
          by auto
    qed
} note lem = this
{  
  assume a \notin O
  from lem [OF this] have a \in - \bigcup ((\lambda(-,-,-,-,O,\cdot)\cdot set ts)
    by (fastforce simp add: in-set-conv-nth)
  with unowned-shared have a \in dom S
    by auto
} then
show ?thesis
  by auto
qed

lemma (in unowned-shared) unowned-shared':
  assumes notin: \forall i < length ts. a \notin owned (ts!i)
  shows a \in dom S
proof -
from notin have a ∈ − ∪ ((λ(·,-,-,-,·,·), ·) · set ts)
    by (force simp add: in-set-conv-nth)
with unowned-shared show ?thesis by blast
qed

lemma unowned-shared-def': unowned-shared \mathcal{S} \text{ ts} = (\forall a. (\forall i < \text{length ts}. a \notin \text{owned (ts!i)}) \rightarrow a \in \text{dom } \mathcal{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 \text{cfg} \equiv let (ts,m,\mathcal{S}) = \text{cfg}
    in unowned-shared \mathcal{S} \text{ ts} ∧
    (\forall i < \text{length ts}. \text{let (p,is,θ,x,D,O,R) } = \text{ts!i in}
      \mathcal{R} = \text{Map.empty })

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

lemma initial-unowned-shared: initial (ts,m,\mathcal{S}) \Longrightarrow \text{unowned-shared } \mathcal{S} \text{ 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 \mathcal{S} m \text{ where c=(ts,m,\mathcal{S}) by (cases c)}
    from initial-empty-rels [OF init [simplified c]]
    have rels-empty: \forall \mathcal{R} \in \text{set (map released ts)}. \mathcal{R} = \text{Map.empty}.
    from initial-unowned-shared [OF init [simplified c]] have unowned-shared \mathcal{S} \text{ ts}
        by auto
    hence us:(\forall a. (\forall i < \text{length (map owned ts)}. a \notin (\text{map owned ts!i)}) \rightarrow a \in \text{dom } \mathcal{S})
        by (simp add:unowned-shared-def')
    
49
fix i p is \( \theta \times D \circ R \)
\[\text{assume}\ i\text{-bound: } i < \text{length ts} \]
\[\text{assume}\ ts\text{-i: } ts!i = (p,\theta,x,D,O,R) \]
\[\text{have}\ \text{map owned ts,map released ts,}i\vdash(is,\theta,m,D,O,S)\sqrt{} \]
\[\text{proof} - \]
\[\text{from}\ \text{safeE [OF safe [simplified c]} i\text{-bound ts-i}\]
\[\text{have}\ \text{map owned ts,}i\vdash(is,\theta,m,D,O,S)\sqrt{}. \]
\[\text{from}\ \text{memop-empty-rels-safe-free-flowing-implies-safe-delayed [OF this rels-empty - us]} i\text{-bound ts-i}\]
\[\text{show} \ ?\text{thesis} \]
\[\text{by simp}\]
\[\text{qed} \}
\[\text{then show} \ ?\text{thesis} \]
\[\text{by} \ (\text{fastforce simp add: c safe-delayed-def})\]
\[\text{qed} \}

locale program-progress = program +
\[\text{assumes}\ \text{progress:} \ \theta\vdash p \rightarrow p' \text{ (}p',is') \implies p' \neq p \lor is' \neq [] \]

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:
\[\text{assumes}\ \text{step:} (ts,m,S) \Rightarrow_d (ts',m',S') \]
\[\text{assumes}\ i\text{-bound: } i < \text{length ts} \]
\[\text{assumes}\ \text{unchanged: } ts!i = ts'!i \]
\[\text{assumes}\ \text{safe-delayed-undo: } \text{safe-delayed (}u\text{-ts},u\text{-m},u\text{-shared)} -- \text{proof should also work with weaker}\ \text{safe-free-flowing} \]
\[\text{assumes}\ \text{leq: } \text{length } u\text{-ts} = \text{length ts} \]
\[\text{assumes}\ \text{others-same: } \forall j < \text{length ts}. j \neq i \rightarrow u\text{-ts}[j] = ts![j] \]
\[\text{assumes}\ u\text{-ts-i: } u\text{-ts}!i = (u\text{-p},u\text{-is},u\text{-temps},u\text{-x},u\text{-dirty},u\text{-owns},u\text{-rels}) \]
\[\text{assumes}\ u\text{-m-other: } \forall a. a \notin u\text{-owns} \rightarrow u\text{-m } a = m\ a \]
\[\text{assumes}\ u\text{-m-shared: } \forall a. a \in u\text{-owns} \rightarrow a \in \text{dom u-shared} \rightarrow u\text{-m } a = m\ a \]
\[\text{assumes}\ u\text{-shared: } \forall a. a \notin u\text{-owns} \rightarrow a \notin \text{owned (}ts![i]) \rightarrow u\text{-shared } a = S\ a \]
\[\text{assumes}\ \text{dist: } \text{simple-ownership-distinct u-ts} \]
\[\text{assumes}\ \text{dist-ts: } \text{simple-ownership-distinct ts} \]
\[\text{shows}\ \exists u\text{-ts}' u\text{-shared}' u\text{-m}'. (u\text{-ts},u\text{-m},u\text{-shared}) \Rightarrow_d (u\text{-ts}',u\text{-m}',u\text{-shared}') \land \]
\[\text{— thread i is unchanged} \]
\[u\text{-ts}'!i = u\text{-ts}![i] \land \]
\[(\forall a \in u\text{-owns}. u\text{-shared}'\ a = u\text{-shared } a) \land \]
\[(\forall a \in u\text{-owns}. S'\ a = S\ a) \land \]
\[(\forall a \in u\text{-owns}. u\text{-m}'\ a = u\text{-m } a) \land \]
\[(\forall a \in u\text{-owns}. m'\ a = m\ a) \land \]

— other threads are simulated
\[(\forall j < \text{length ts}. j \neq i \rightarrow u\text{-ts}'[j] = ts'![j] ) \land \]
\[(\forall a. a \notin u\text{-owns} \rightarrow a \notin \text{owned (}ts![i]) \rightarrow u\text{-shared}'\ a = S'\ a) \land \]
\[(\forall a. a \notin u\text{-owns} \rightarrow u\text{-m}'\ a = m'\ a) \land \]
\[\text{proof} - \]
\[\text{interpret}\ \text{direct-computation:} \]
computation direct-memop-step empty-storebuffer-step program-step λ p p’ is sb. sb .
from dist interpret simple-ownership-distinct u-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, sb, D, O, R)] and
  S’; S’ = S and
  m; m’ = m and
j-bound: j < length ts and
ts-j: ts|j = (p, is, θ, sb, D, O, R) and
prog-step: θ ⊢ p → p (p’, is’)
  by auto

from progress [OF prog-step] i-bound unchanged ts-j ts’
have neq-j-i: j ≠ i
  by auto

from others-same [rule-format, OF j-bound neq-j-i] ts-j
have u-ts-j: u-ts|j = (p, is, θ, 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 direct-computation.Program [OF j-bound’ u-ts-j prog-step]
have ustep: (u-ts, u-m, u-shared) ⇒d (u-ts|j := (p’, is’@is’, θ, sb, D, O, R)), u-m, u-shared). show ?thesis
  apply –
  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 same u-ts-j
  apply (auto simp add: nth-list-update ts’ S’ m’)
done
next
case (Memop j p is θ sb D O R is’ θ’ sb’ D’ O’ R’)
them obtain
  ts’; ts’ = ts[j := (p, is’, θ’, sb’, D’, O’, R’) ] and
j-bound: j < length ts and
ts-j: ts|j = (p, is, θ, sb, 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 mem-step i-bound unchanged ts-j

51
have neq-j-i: \(j \neq i\)
  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,\(\emptyset\),sb,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 safe-delayedE [OF safe-delayed-undo j-bound′ u-ts-j]
have safe-j: map owned u-ts,map released u-ts[j⇒(is, \(\emptyset\), 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: \(\mathcal{O} \cap \text{u-owns} = \{\}\).
from mem-step show ?thesis
proof (cases)
case (Read volatile a t)
  then obtain
    is: is = Read volatile a t \# is′ and
    \(\emptyset′\): \(\emptyset′ = \emptyset(t \mapsto m a)\) 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′ = \(\emptyset′\) 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]

52
show ?thesis.

qed

note Read′ = direct-memop-step.Read [of volatile a t is′ θ sb u-m D O R u-shared]
from direct-computation.Memop [OF j-bound′ u-ts-j, simplified is, OF Read′]
have ustep: (u-ts, u-m, u-shared) ⇒d (u-ts[j := (p, is′, θ(t → u-m a), sb, 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
by (auto simp add: nth-list-update ts′ eqs′ u-m-a-eq)

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 safe-j [simplified is]
obtain
  owned: a ∈ O and unshared: a /∈ 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 /∈ u-owns
  by auto
note Write′ = direct-memop-step.WriteNonVolatile [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, D, O, R)], u-m (a := f θ), 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 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-unowned-others: ∀ k < length u-ts. j ≠ k −→ A ∩ (map owned u-ts!k ∪ dom (map released u-ts!k)) = {}
  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) ⇒ₜ (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 ⊕ₜ 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
  assume a-u-owns: a /∈ u-owns
  assume a-u-owns-orig: a /∈ owned (ts!i)

54
from u-shared [rule-format, OF a-u-owns a-u-owns-orig]
have (u-shared ⊕ W R ⊕ A L) a = (S ⊕ W R ⊕ 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 ∈ u-owns

  have (S ⊕ W R ⊕ A L) a = S 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 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 Fence
then obtain
  is: is = Fence # is′ and
  û!: û′ = û 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′ = û′ sb′ m′ D′ O′ R′ S′
note Fence′ = direct-memop-step.Fence [of is′ û sb u-m D O R u-shared]
from direct-computation.Memop [OF j-bound′ u-ts-j, simplified is, OF Fence′]
have ustep: (u-ts, u-m, u-shared) ⇒d (u-ts[j := (p, is′, û, 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')

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
\[ \vartheta' : \vartheta' = \vartheta(t \mapsto m a) \] 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
\[ cond: \neg cond (t \mapsto m a) \]
by auto
note eqs' = \[ \vartheta' sb' m' D' O' R' S' \]
from safe-j [simplified is] owns-u-owns u-ts-i i-bound' neq-j-i
obtain
access-cond: a \notin u-owns \lor (a \in dom u-shared \land a \in 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': \neg cond (t \mapsto u-m a))
by auto
note RMWReadOnly' = direct-memop-step.RMWReadOnly [of cond \vartheta 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) \Rightarrow \( u-t s[j := (p, is', \vartheta(t \mapsto u-m a), sb, False, O, Map.empty)], u-m, u-shared)\).
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
\[ \vartheta' : \vartheta' = \vartheta(t \mapsto ret (m a) (f (\vartheta(t \mapsto m a)))) \] and
\[ sb' : sb' = sb \] and
\[ m' : m' = m(a := f (\vartheta(t \mapsto m a))) \] and
\[ D': D' = \text{False and} \]
\[ O': O' = O \cup A - R \quad \text{and} \]
\[ R': R' = \text{Map.empty and} \]
\[ S': S' = S \oplus W R \ominus A L \quad \text{and} \]
cond: \( \text{cond} (\emptyset (t \mapsto m a)) \)
by auto

**note** eqs\' = \( \theta' \) 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 \( \notin \) u-owns \( \lor \) (a \( \in \) dom u-shared \( \land \) a \( \in \) 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': \( \text{cond} (\emptyset (t \mapsto u-m a)) \)
by auto
from safe-j [simplified is] cond' **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 \( \subseteq \) dom u-shared \( \cup \) O and L-A: L \( \subseteq \) A and R-owns: R \( \subseteq \) O and A-R: A \( \cap \) R = \{\} \( \text{and} \)
A-unowned-others: \( \forall k < \text{length u-ts}. j \neq k \rightarrow A \cap (\text{map owned u-ts!k} \cup \text{dom (map released u-ts!k)}) = \{\} \) \( \text{and} \)
a-not-ro: a \( \notin \) read-only u-shared
by cases auto

**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,
OF cond'
from direct-computation.Memop [OF j-bound' u-ts-j, simplified is, OF Write']
**have** ustep: (u-ts, u-m, u-shared) \( \Rightarrow_d \)
(u-ts[j := (p, is', \emptyset (t \mapsto \text{ret} (u-m a)) (f (\emptyset (t \mapsto u-m a))))), sb, False, O \cup A - R, Map.empty)),
\( u-m(a := f (\emptyset (t \mapsto u-m a)))) \), u-shared \( \oplus W R \ominus A L \).

from A-unowned-others [rule-format, OF i-bound' neq-j-i] u-ts-i i-bound'
**have** A-u-owns: A \( \cap \) u-owns = \{\} by auto
{}
fix a
**assume** a-u-owns: a \( \in \) u-owns
**have** (u-shared \( \oplus W R \ominus A L\)) 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 \( \notin \) u-owns by auto
\{ 
  \textit{fix} a 
  \textit{assume} \ a-u-owns: a \notin u-owns 
  \textit{assume} \ a-u-owns-orig: a \notin owned (ts!i) 
  \textit{from} \ u-shared [\textit{rule-format}, \textit{OF} a-u-owns this] 
  \textit{have} \ (u-shared \oplus_W R \ominus_A L) \ a = (S \oplus_W R \ominus_A L) \ a 
  \textit{using} \ R-owns A-R L-A A-u-owns owns-u-owns 
  \quad \textit{by} \ (\textit{auto simp add: restrict-shared-def augment-shared-def split: option.splits}) 
\} 
\textit{note} \ u-unowned-shared = this 
\{ 
  \textit{fix} a 
  \textit{assume} \ a-u-owns: a \in u-owns 
  \textit{have} \ (S \oplus_W R \ominus_A L) \ a = S \ a 
  \textit{using} \ R-owns A-R L-A A-u-owns owns-u-owns 
  \quad \textit{by} \ (\textit{auto simp add: restrict-shared-def augment-shared-def split: option.splits}) 
\} 
\textit{note} S'-shared = this 
\textit{show} ?thesis 
  \textit{apply} – 
  \textit{apply} \ (\textit{rule exI}) 
  \textit{apply} \ (\textit{rule exI}) 
  \textit{apply} \ (\textit{rule exI}) 
  \textit{apply} \ (\textit{rule conjI}) 
  \textit{apply} \ (\textit{rule ustep}) 
  \quad \textit{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 
  \textit{apply} \ (\textit{auto simp add: nth-list-update ts' eqs'}) 
\textit{done} 
\textit{next} 
\textit{case} (\textit{Ghost} A L R W) 
\textit{then obtain} 
  is: is = Ghost A L R W \ # is' \textit{and} 
  \emptyset', \emptyset = \emptyset \textit{and} 
  sb': sb'=sb \textit{and} 
  m': m'=m \textit{and} 
  D': D'=D \textit{and} 
  O': O'=O \cup A - R \textit{and} 
  R': R'=\textit{augment-rels} (\textit{dom} S) R R \textit{and} 
  S': S'=S \oplus_W R \ominus_A L 
  \textit{by} \ \textit{auto} 
\textit{note} eqs' = \emptyset' sb' m' D' O' R' S' 
\textit{from} safe-j [\textit{simplified is}] 
\textit{obtain} 
  A: A \subseteq \textit{dom} u-shared \cup O \textit{and} L-A: L \subseteq A \textit{and} R-owns: R \subseteq O \textit{and} A-R: A \cap R 
  = \{\} \textit{and} 
  \textit{A-unowned-others}: \forall k < \textit{length} u-ts. j\neq k \rightarrow A \cap (\textit{map} owned u-ts!k \cup \textit{dom} (\textit{map} released u-ts!k)) = \{\} 
\}
by cases auto

note Ghost' = direct-memop-step.Ghost [of A L R W is' \varnothing sb u-m D \mathcal{O} R u-shared]
from direct-computation.Memop [OF j-bound' u-ts-j, simplified is, OF Ghost']
have ustep: (u-ts, u-m, u-shared) \rightarrow_d 
    (u-ts[j] := (p, is', \varnothing, sb, D, \mathcal{O} \cup A - R, augment-rels (dom u-shared) R R 
)), u-m,
    u-shared \oplus_W R \ominus_A L).

from A-unowned-others [rule-format, OF i-bound' neq-j-i] u-ts-i i-bound'
have A-u-owns: A \cap u-owns = {} by auto
{
  fix a
  assume a-u-owns: a \in u-owns
  have (u-shared \oplus_W R \ominus_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
{
  fix a
  assume a-u-owns: a \notin u-owns
  assume 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
  by (auto simp add: restrict-shared-def augment-shared-def split: option.splits)
}
note S'-shared = this

from dist-ts
from dist-ts-inter.simple-ownership-distinct [OF j-bound i-bound neq-j-i ts-j]
have \mathcal{O} \cap owned (ts!i) = {}
apply (cases ts!i)
apply fastforce+
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 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 ∅ 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) ⇒ₜ (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 λp p′ is sb. sb .
from dist interpret simple-ownership-distinct ts .
from step
show ?thesis
proof (cases)
case (Program j p is ∅ sb D O R p′ is′)
then obtain
  ts′: ts′ = ts[j := (p′,is′@is′,∅,sb,D,O,R)] and
  S′: S′⊆ S and
  m′: m′ = m and
  j-bound: j < length ts and
  ts-j: ts[j] = (p,is,∅,sb,D,O,R) and
  prog-step: ∅ ⊢ p →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 D OR is' \theta' sb' D' O' R')
  then obtain
    ts': ts' = ts[j:=(p,is',\theta',sb',D',O',R')] and
    j-bound: j < length ts and
    ts-j: tsj = (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 safe-delayedE [OF safe j-bound ts-j]
have safe-j: map owned ts,map released ts,j \vdash (is, \theta, m, D, O, S) \sqrt.
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 \theta) and
    D': D'=D and
\( O' : O' = O \) and
\( R' : R' = R \) and
\( S' : S' = S \)
by auto
note eqs' = \( \emptyset' \) 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 (WriteVolatile a D f A L R W)
then obtain
is: is = Write True a (D, f) A L R W # is' and
\( \emptyset' : \emptyset' = \emptyset \) and
sb': sb' = sb and
m': m' = m(a := f) and
\( D' : D' = \text{True} \) and
\( O' : O' = O \cup A - R \) and
\( R' : R' = \text{Map.empty} \) 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-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 \( \notin \) read-only S
by cases auto
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 Fence
then obtain
is: is = Fence # is' and
\( \emptyset' : \emptyset' = \emptyset \) and
sb': sb' = sb and
m': m' = m and
\( D' : D' = \text{False} \) and
\( O' : O' = O \) and
\( R' : R' = \text{Map.empty} \) and
\( S' : S' = S \)

by auto

**note** eqs' = \( \emptyset' sb' m' D' O' R' S' \)

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 (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
\( \emptyset' : \emptyset' = \emptyset(t \mapsto m a) \) 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: \( \neg \text{cond} (\emptyset(t \mapsto m a)) \)
by auto

**note** eqs' = \( \emptyset' sb' m' D' O' R' S' \)

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 (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
\( \emptyset' : \emptyset' = \emptyset(t \mapsto \text{ret} (m a) (f (\emptyset(t \mapsto m a)))) \) and
sb': sb' = sb and
m': m' = m(a := f (\emptyset(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 (\( \emptyset(t \mapsto m a) \))
by auto

**note** eqs' = \( \emptyset' sb' m' D' O' R' S' \)

from safe-j [simplified is] cond
obtain

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

63

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 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 (Ghost A L R W)
then obtain
  is: is = Ghost A L R W ≠ is' and
  φ; φ' = φ and
  sb'; sb'=sb and
  m'; m'=m and
  D'; D'=D and
  O'; O'=O ∪ A - R and
  R'; R'=augment-rels (dom S) R R and
  S'; S'=S ⊕_w R ⊕_A L
  by auto
note eqs' = φ' sb' m' D' O' R' S'

from safe-j [simplified is]
obtain

and
A-unowned-others: ∀k < length ts. j≠k → A ∩ (map owned ts!k ∪ dom (map released ts!k)) = {} by cases auto

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
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

64
theorem (in program) safe-step-preserves-read-only-unowned:
assumes step: (ts,m,S) ⇒ d (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 dist 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,sb,D,O,R)] and
S': S'=S and
m': m'=m and
j-bound: j < length ts and
ts-j: ts!j = (p,is,θ,sb,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',sb',D',O',R')] and
j-bound: j < length ts and
ts-j: ts!j = (p,is,θ,sb,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
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′ = ǂ′ 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
  ǂ′: ǂ′ = ǂ and
  sb′: sb′=sb and
  m′: m′=m(a:=f ǂ) and
D′: D′=D and
O′: O′=O and
R′: R′=R and
S′: S′=S
by auto
note eqs′ = ǂ′ 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
  ǂ′: ǂ′ = ǂ and
  sb′: sb′=sb and
  m′: m′=m(a:=f ǂ) and
D′: D′=True and
O′: O′=O ∪ A − R and
R′: R′=Map.empty and
S′: S′⊕W R ⊖A L
by auto
note eqs′ = ǂ′ sb′ m′ D′ O′ R′ S′
from safe-j [simplified is]
obtain
a-unowned-others: \( \forall k < \text{length } ts \ j \neq k \rightarrow a \notin (\text{map } \text{owned } ts!k \cup \text{dom } (\text{map } \text{released } ts!k)) \) 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: \( \forall k < \text{length } ts \ j \neq k \rightarrow A \cap (\text{map } \text{owned } ts!k \cup \text{dom } (\text{map } \text{released } ts!k)) = \{\} \) and

a-not-ro: \( a \notin \text{read-only } S \)

by cases auto

show read-only-unowned \( S' \) \( ts' \)

proof (unfold-locales)

fix i \( p_i \) \( is_i \) \( O_i \) \( R_i \) \( D_i \) \( \theta_i \) \( sb_i \)

assume i-bound: \( i < \text{length } ts' \)

assume \( ts'{-i} \): \( ts'{-i} = (p_i, is_i, \theta_i, sb_i, D_i, O_i, R_i) \)

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' \text{read-only-def} \text{augment-shared-def} \text{restrict-shared-def} domIff split: option.splits)

next

case False

from simple-ownership-distinct \[ OF \ j-bound - False \ [\text{symmetric}] ts-j \] \( ts'{-i} \) 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]

show ?thesis

by (auto simp add: eqs' ts' \text{read-only-def} \text{augment-shared-def} \text{restrict-shared-def} domIff split: option.splits)

qed

qed

next

case Fence

then obtain

is: \( is = \text{Fence} \ # \ is' \) and

\( \theta': \theta = \theta \) and

\( sb': sb' = sb \) and

\( m': m = m \) and

\( D': D = \text{False} \) and

\( O': O = O \) and

\( R': R = \text{Map.empty} \) and

\( S': S = S' \)

by auto

note eqs' = \( \theta' sb' m' D' O' R' S' \)

from read-only-unowned \[ OF \ j-bound ts-j \]

show read-only-unowned \( S' \) \( ts' \)

67
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
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 (φ(t → m a)) 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 (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
sb': sb' = sb and
m': m' = m(a := f (θ(t → m a))) and
D': D' = False and
O': O' = O ∪ A − R and
R': R' = Map.empty and
S': S' = S ⊕ W R ⊖ A L and
cond: cond (φ(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 read-only-unowned S' ts'
proof (unfold-locales)
fix i p_1 i_1 o_1 d_1 \theta_i s_b_i
assume i-bound: i < length ts'
assume ts'\:i: ts'\:i = (p_1, i_1, \theta_i, s_b_i, d_1, o_1, r_1)
show o_1 \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_1 \cap o_1 = \{\}
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_1 i_1 o_1 s_b_1 d_1 o_1 r_1]
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
\theta': \theta' = \emptyset and
s_b': s_b' = s_b and
m': m' = m and
d': d' = d and
o': o' = o \cup A - R and
r': r' = \text{augment-rels (dom } S\text{) R R and}
S': S' = S \oplus W R \ominus A L
by auto
note eqs' = \emptyset s_b' m' d' o' r' S'
from safe-j [simplified is]
obtain
A: A \subseteq \text{dom } S \cup o \text{ 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'
proof (unfold-locales)
fix i p_i i_s o_i r_i d_i \theta_i s_b_i
assume i-bound: i < length ts’
assume ts’-i: ts’\i = (p_i, is_i, \theta_i, sb_i, D_i, O_i, R_i)
show O_i \cap 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 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 (StoreBuffer - p is \theta 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-unowned-shared:
  assumes step: (ts,m,S) \Rightarrow_d (ts’,m’,S’)
  assumes safe: safe-delayed (ts,m,S)
  assumes dist: simple-ownership-distinct ts
  assumes unowned-shared: unowned-shared S
  shows unowned-shared S’ ts
proof –
  interpret direct-computation:
    computation direct-memop-step empty-storebuffer-step program-step \lambda p 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 \theta sb D O R p’ is’)
    then obtain
      ts’: ts’ = ts[j:=(p’,is@is’,\theta, sb, D,O, R)]
  end
$S': S' = S$ 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

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

next
case (Memop $j p$ is $\theta$ sb $D$ $O$ $R$ is $\theta'$ sb $D'$ $O'$ $R'$)
them obtain
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: $(is, \theta, sb, m, D, O, S) \rightarrow (is', \theta', sb', m', D', O', R', S')$
by auto

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

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 (WriteNonVolatile $a D f A L R W$)
them obtain
is: $is = \text{Write False } a (D, f) A L R W \# is'$ and
$\theta': \theta' = \theta$ and
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
    ∅'': ∅' = ∅ and
    sb': sb' = sb and
    m': m' = m(a:=f ∅) 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' = ∅' sb' m' D' O' R' S'

from safe-j [simplified is]
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

case Fence
then obtain
  is: is = Fence # is' and
  θ'; θ' = θ 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' = θ' 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
  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
  cond: ¬ cond (θ(t ↦→ m a))
by auto
note eqs' = θ' 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
  θ; θ' = θ(t↦→ ret (m a) (f (θ(t ↔ m a)))) and
  sb; sb'=sb and
  m; m'=m(a := f (θ(t ↔ m a))) and
  D; D'=False and

\[ O' : O' = O \cup A - R \quad \text{and} \]
\[ R' : R' = \text{Map.empty} \quad \text{and} \]
\[ S' : S' = S \oplus_W R \ominus_A L \quad \text{and} \]
\[ \text{cond: cond} (\emptyset (t \mapsto m a)) \]
\[ \text{by auto} \]
\[ \text{note eqs'} = \emptyset 'sb' m'D'O'R'S' \]
\[ \text{from safe-j [simplified is] cond} \]
\[ \text{obtain} \]
\[ \quad \text{a-unowned-others: } \forall k < \text{length ts. } j \neq k \rightarrow a \notin (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) \quad \text{and} \]
\[ \quad A: A \subseteq \text{dom } S \cup O \quad \text{and} \]
\[ \quad \text{L-A: } L \subseteq A \quad \text{and} \quad \text{R-owns: } R \subseteq O \quad \text{and} \]
\[ \quad \text{A-R: } A \cap R = \{\} \]
\[ \text{and} \]
\[ \quad \text{A-unowned-others: } \forall k < \text{length ts. } j \neq k \rightarrow A \cap (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) = \{\} \quad \text{and} \]
\[ \quad \text{a-not-ro: } a \notin \text{read-only } S \]
\[ \quad \text{by cases auto} \]
\[ \quad \text{show unowned-shared } S' \text{ ts'} \]
\[ \quad \text{apply (clarsimp simp add: unowned-shared-def')} \]
\[ \quad \text{using A R-owns L-A A-R \text{A-unowned-others ts-j j-bound}} \]
\[ \quad \text{apply (auto simp add: } S' \text{ ts'} O') \]
\[ \quad \text{apply (rule unowned-shared')} \]
\[ \quad \text{apply clarsimp} \]
\[ \quad \text{apply (drule-tac } x=i \text{ in spec)} \]
\[ \quad \text{apply (case-tac } i=j) \]
\[ \quad \text{apply clarsimp} \]
\[ \quad \text{apply clarsimp} \]
\[ \quad \text{apply (drule-tac } x=j \text{ in spec)} \]
\[ \quad \text{apply auto} \]
\[ \quad \text{done} \]
\[ \text{next} \]
\[ \text{case (Ghost A L R W)} \]
\[ \text{then obtain} \]
\[ \quad \text{is: is = Ghost A L R W \# is'} \quad \text{and} \]
\[ \quad \emptyset '; \emptyset' = \emptyset \quad \text{and} \]
\[ \quad \text{sb': sb'=sb \quad and} \]
\[ \quad m': m'=m \quad \text{and} \]
\[ \quad \text{D': D'=D \quad and} \]
\[ \quad O': O'=O \cup A - R \quad \text{and} \]
\[ \quad R': R'=\text{augment-rels (dom } S) \text{ R R and} \]
\[ \quad S': S'=S \oplus_W R \ominus_A L \]
\[ \quad \text{by auto} \]
\[ \text{note eqs'} = \emptyset 'sb' m'D'O'R'S' \]
\[ \text{from safe-j [simplified is]} \]
\[ \text{obtain} \]
\[ \quad A: A \subseteq \text{dom } S \cup O \quad \text{and} \quad \text{L-A: } L \subseteq A \quad \text{and} \quad \text{R-owns: } R \subseteq O \quad \text{and} \quad \text{A-R: } A \cap R = \{\} \]
\[ \quad \text{and} \]
\[ \quad \text{A-unowned-others: } \forall k < \text{length ts. } j \neq k \rightarrow A \cap (\text{map owned ts!k} \cup \text{dom (map released ts!k)}) = \{\} \]
by cases auto
show unowned-shared $S'$ ts'
apply (clarsimp simp add: unowned-shared-def)
using A R-owns L-A 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
apply clarsimp
apply (drule-tac x=j in spec)
apply auto
done
qed

next
case (StoreBuffer - p is \emptyset \ D \ O \ R \ sb' \ O' \ R')
hence False
by (auto simp add: empty-storebuffer-step-def)
thus ?thesis ..
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::nat — starting index
fixes k::nat — steps

assumes step: $\forall l. l < k =\Rightarrow_d c \ (n+l) \Rightarrow_d c \ (n + (Suc l))$

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: $(\forall x < Suc k. \ P \ x) = (P \ k \land (\forall x < k. \ P \ x))$
apply rule
apply clarsimp
apply clarsimp
apply clarsimp
apply (case-tac x = k)
apply auto
done

lemma split-le-Suc: $(\forall x \leq Suc k. \ P \ x) = (P \ (Suc k) \land (\forall x \leq k. \ P \ x))$
apply rule
apply clarsimp
apply clarsimp
apply (case-tac x = Suc k)
apply auto
done

lemma (in program) steps-to-trace:
assumes steps: x ⇒ d * y
shows ∃ c k. trace c 0 k ∧ c 0 = x ∧ 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 ⇒ d * y by fact
have last: y ⇒ d z by fact
from step.hyps obtain c k where
  trace: trace c 0 k and c-0: c 0 = x and c-k: 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 (subgoal-tac l=k)
  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:
∀ l x. trace c n k ⇒ l ≤ k ⇒ x ≤ k ⇒ length (fst (c (n + l))) = length (fst (c (n + x)))
proof (induct k)
case 0
  thus ?case by auto
next
case (Suc k)
then obtain trace-suc: trace c n (Suc k) and
l-suc: l ≤ Suc k and
x-suc: x ≤ 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 + (Suc k))
  by (clarsimp simp add: program-trace-def)
obtain ts \( \mathcal{S} \) m where c-k: c (n + k) = (ts, m, \( \mathcal{S} \)) by (cases c (n + k))
obtain ts' \( \mathcal{S}' \) m' where c-suc-k: c (n + (Suc k)) = (ts', m', \( \mathcal{S}' \)) by (cases c (n + (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 = Suc k)
  case True
  note l-suc = this
  show ?thesis
  proof (cases x = 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 = 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

lemma (in program) trace-preserves-simple-ownership-distinct:
assumes dist: simple-ownership-distinct (fst (c n))
shows \( \forall l. \text{trace } c \ n \ k \implies (\forall x < k. \text{safe-delayed } (c (n + x))) \implies l \leq k \implies \text{simple-ownership-distinct } (\text{fst } (c (n + l))) \)

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: \( c (n + k) \Rightarrow_d c (n + (\text{Suc k})) \)
by (clarsimp simp add: program-trace-def)

obtain ts \( \mathcal{S} \) m where c-k: \( c (n + k) = (ts, m, \mathcal{S}) \) by (cases \( c (n + k) \))
obtain ts' \( \mathcal{S}' \) m' where c-suc-k: \( c (n + (\text{Suc k})) = (ts', m', \mathcal{S}') \) 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: \( \text{safe-delayed } (ts, m, \mathcal{S}) \)
by (auto simp add: split-le-Suc)
from Suc.hyps [OF trace-k safe-up-k]
have hyp: \( \forall l \leq k. \text{simple-ownership-distinct } (\text{fst } (c (n + l))) \)
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 \( \forall l. \text{trace } c \ n \ k \implies (\forall x < k. \text{safe-delayed } (c (n + x))) \implies l \leq k \implies \text{read-only-unowned } (\text{snd } (\text{snd } (c (n + l)))) (\text{fst } (c (n + l))) \)
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: \( \text{safe-delayed} (ts, m, S) \)
    by (auto simp add: split-le-Suc)
  from Suc.hyps [OF trace-k safe-up-k]
  have hyp: \( \forall l \leq k. \text{read-only-unowned} (\text{snd} (\text{snd} (c (n + l)))) (\text{fst} (c (n + l))) \)
    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} ts by simp
  from safe-step-preserves-read-only-unowned [OF last-step[simplified c-k c-suc-k] safe-k dist' ro']
  have 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: \text{unowned-shared} \((\text{snd} \ (\text{snd} \ (c \ n)))\) \((\text{fst} \ (c \ n))\)
  shows \( \forall l. \text{trace} \ c \ n \ k \Rightarrow (\forall x < k. \text{safe-delayed} \ (c \ (n + x))) \Rightarrow \)
\[ l \leq k \implies \text{unowned-shared (snd (snd (c (n + l)))) (fst (c (n + l)))} \]

**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 \( \mathcal{S} \) m where c-k: \( c (n + k) = (ts, m, \mathcal{S}) \) by (cases c (n + k))

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

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

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

have hyp: \( \forall 1 \leq k. \, \text{unowned-shared (snd (snd (c (n + l)))) (fst (c (n + l)))} \)

by simp

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

have ro’: unowned-shared \( \mathcal{S} \) 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 \( \mathcal{S}' \) 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: \( c \cdot n = (\alpha, m, \sigma) \)
assumes unchanged: \( \forall l \leq k \cdot (\forall \alpha, \beta, \gamma \cdot c \cdot (n + l) = (\alpha, \beta, \gamma) \rightarrow \alpha = \beta \land \gamma) \)
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: \( \forall j < length ts \cdot j \neq i \rightarrow u-ts!j = ts!j \)
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: \( \forall a. a \notin u-owns \rightarrow u-m a = m a \)
assumes u-m-shared: \( \forall a. a \in u-owns \rightarrow a \in dom u-shared \rightarrow u-m a = m a \)
assumes u-shared: \( \forall a. a \notin u-owns \rightarrow a \notin owned (ts!i) \rightarrow u-shared a = S a \)
assumes dist: simple-ownership-distinct u-ts
assumes dist-ts: simple-ownership-distinct ts
assumes safe-orig: \( \forall x. x < k \rightarrow safe-delayed (c \cdot (n + x)) \)
shows \( \exists c' \cdot 1 \leq k \land trace c' \cdot n \land \)

\[
\begin{align*}
& c' \cdot n = (u-ts, u-m, u-shared) \\
& (\forall x \leq 1 \cdot length (fst (c' \cdot (n + x))) = length (fst (c \cdot (n + x)))) \\
& (\forall x < 1 \cdot safe-delayed (c' \cdot (n + x))) \\
& (1 < k \rightarrow \neg safe-delayed (c' \cdot (n + 1))) \\
& (\forall x \leq 1 \cdot \forall ts' \cdot S' \cdot m' \cdot c' \cdot (n + x) = (ts', m', S') \\
& \quad \rightarrow ts'!i = u-ts!i \\
& \quad \land (\forall a \in u-owns. S' a = u-shared a) \\
& \quad \land (\forall a \in u-owns. S' a = S a) \\
& \quad \land (\forall a \in u-owns. m' a = u-m a) \\
& \quad \land (\forall a \in u-owns. m' a = m a)) \\
& (\forall x \leq 1 \cdot \forall ts' \cdot S' \cdot m' \cdot c' \cdot (n + x) = (ts', m', S') \\
& \quad \rightarrow c' \cdot (n + x) = (ts', m', S') \\
& \quad \rightarrow (\forall j < length ts' \cdot j \neq i \rightarrow ts'!j = ts!j) \\
& \quad \land (\forall a. a \notin u-owns \rightarrow a \notin owned (ts!i) \rightarrow S' a = S a) \\
& \quad \land (\forall a. a \notin u-owns \rightarrow m' a = m a))
\end{align*}
\]

using steps unchanged safe-orig

proof (induct k)

\begin{itemize}
  \item case 0
    \begin{itemize}
      \item show ?case
        \begin{itemize}
          \item apply (rule-tac x=\lambda l. (u-ts, u-m, u-shared) in exI)
          \item apply (rule-tac x=0 in exI)
          \item thm c-n
          \item apply (simp add: c-n)
          \item apply (clarsimp simp add: 0 leq others-same u-m-other u-shared)
        \end{itemize}
      \end{itemize}
    \end{itemize}

  \item done
\end{itemize}

next

\begin{itemize}
  \item case (Suc k)
  \begin{itemize}
    \item then obtain
      \begin{itemize}
        \item trace-suc: trace c n (Suc k) and
        \item unchanged-suc: \( \forall l \leq Suc k \cdot \forall ts' \cdot S' \cdot m' \cdot c \cdot (n + l) = (ts', m', S') \rightarrow ts'!i = ts!i \land \)
      \end{itemize}
    \end{itemize}
  \end{itemize}

81
safe-origin: \( \forall x < k \). safe-delayed \((c (n + x))\)

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-k: trace \( c \ n \ k \) and
- last-step: \( c \ (n + k) \rightarrow_d c \ (n + (Suc k)) \)

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

**from** unchanged-suc obtain
- unchanged-k: \( \forall l \leq k \). \( \forall ts \ S \ m \ l \). \( c \ (n + l) = (ts, m, S) \rightarrow ts ! i = ts ! i \) and
- unchanged-suc-k: \( \forall ts \ S \ m \ l \). \( c \ (n + (Suc k)) = (ts, m, S) \rightarrow ts ! i = ts ! i \)

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-origin] obtain \( c' \ l \) where
- l-k: \( l \leq k \) and
- trace-c'-l: trace \( c' \ n \ l \) and
- safe-l: \( \forall x < l \). safe-delayed \((c' (n + x))\) and
- unsafe-l: \( l < k \rightarrow \neg \) safe-delayed \((c' (n + l))\) and
- c'-n: \( c' \ n = (u-ts, u-m, u-shared) \) and
- leq-l: \( \forall x \leq l \). length \((fst (c' (n + x)))\) = length \((fst (c (n + x)))\) and
- unchanged-i: \( \forall x \leq l \). \( \forall ts \ x \ S \ x \ m \ x \ ts \ x \ x' \ S \ x \ x' \ m \ x \). \( c \ (n + x) = (ts, m, S) \rightarrow c' (n + x) = (ts, m, S) \rightarrow ts \ x' != i \rightarrow ts \ x' ! i = ts ! i \) ∧
  \( \forall a \in u-owns. S \ x' a = u-shared a \) ∧
  \( \forall a \in u-owns. S \ a = S a \) ∧
  \( \forall a \in u-owns. m \ x a = u-m a \) ∧
  \( \forall a \in u-owns. m \ a a = m a \) and
- sim: \( \forall x \leq l \). \( \forall ts \ x \ S \ x \ m \ x \ ts \ x' \ S \ x' \ m \ x' \).
  \( c \ (n + x) = (ts, m, S) \rightarrow c' (n + x) = (ts, m, S) \rightarrow\)
  \( (\forall j < length ts. j \neq i \rightarrow ts \ x' ! j = ts \ x ! j) \) ∧
  \( \forall a. a \notin u-owns \rightarrow a \notin owned (ts!i) \rightarrow S \ x' a = S \ a a \) ∧
  \( \forall a. a \notin u-owns \rightarrow m \ x' a = m \ a a \) and

by auto
show ?case
proof (cases \( l < k \))
  case True
  with True 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=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_k ! i = u-ts ! i and
unchanged-shared: ∀a∈u-owns. u-shared_k a = u-shared a  \textbf{and}
unchanged-shared-orig: ∀a∈u-owns. S_k a = S a  \textbf{and}
unchanged-owns: ∀a∈u-owns. u-m_k a = u-m a  \textbf{and}
unchanged-owns-orig: ∀a∈u-owns. m_k a = m a
\textbf{by fastforce}

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

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

\textbf{from} safe-l l-k safe-k
\textbf{have} safe-up-k: ∀x<k. safe-delayed (c′ (n + x))
\textbf{apply}clarsimp
\textbf{done}
\textbf{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
\textbf{have} dist′: simple-ownership-distinct u-ts_k
\textbf{by} simp

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

\textbf{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′
\textbf{obtain} u-ts′/u-shared′/u-m′/where
\textbf{step'}: (u-ts_k, u-m_k, u-shared_k) ⇒_d (u-ts′, u-m′, u-shared′)  \textbf{and}
ts-eq': u-ts′ ! i = u-ts_k ! i  \textbf{and}
unchanged-shared′: (∀a∈u-owns. u-shared′ a = u-shared_k a)  \textbf{and}
unchanged-shared-orig′: (∀a∈u-owns. S_k′ a = S_k a)  \textbf{and}
unchanged-owns′: (∀a∈u-owns. u-m′ a = u-m_k a)  \textbf{and}
unchanged-owns-orig′: (∀a∈u-owns. m_k′ a = m_k a)  \textbf{and}

84
sim-ts': (∀ j<length ts. j ≠ i → u-ts' ! j = ts_k' ! j) and
sim-shared': (∀ a. a /∈ u-owns → a /∈ owned (ts_k ! i) → u-shared' a = S_k' a) and
sim-m': (∀ a. a /∈ u-owns → u-m' a = m_k' a)

by auto

define c'' where c'' == λ l. if l ≤ n + k then c' l else (u-ts' l, u-m' l, u-shared')

have [simp]: ∀ x ≤ n + k. c'' x = c' x
  by (auto simp add: c''-def)

have [simp]: c'' (Suc (n + k)) = (u-ts', u-m', u-shared')
  by (auto simp add: c''-def)

from trace-c' l l-k step' c' k 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': length ts_k' = length ts_k.

with direct-computation.step-preserves-length-ts [OF step'] leq
have leq': length u-ts' = length ts_k
  by simp

show ?thesis
  apply (rule-tac x=c'' in exI)
  apply (rule-tac x=Suc (n + k) in exI)
  using safe-l l-k unchanged-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
    unchanged-shared' unchanged-shared unchanged-shared-orig
    unchanged-shared-orig'
    unchanged-owns' unchanged-owns
    unchanged-owns-orig' unchanged-owns-orig )
  done
qed
qed
qed

locale program-safe-reach-uppto = program +
  fixes n fixes safe fixes c_0
  assumes safe-config: [k ≤ n; trace c 0 k; c 0 = c_0; 1 ≤ k ] ⇒ safe (c 1)

abbreviation (in program)
  safe-reach-uppto ≡ program-safe-reach-uppto program-step

lemma (in program) safe-reach-uppto-le:
  assumes safe: safe-reach-uppto n safe c_0
  assumes m-n: m ≤ n

85
shows safe-reach-upto m safe c₀
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
          (∀l ≤ 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 < l −→ 1 ≤ 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 ⇒₄ 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: \( (\forall 1 \leq k. \text{fst} (c l) ! i = \text{fst} (c k) ! i) \)

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: \( (\forall 1 > \text{last}. 1 \leq k \rightarrow \text{fst} (c l) ! i = \text{fst} (c k) ! i) \)
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 (case-tac l=Suc k)
apply clarsimp
apply clarsimp
done
}

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


lemma (in program) sequence-traces:
assumes trace1: trace c \_ 0 k
assumes trace2: trace c \_ m l
assumes seq: c \_ m = c \_ k
assumes c-def: \( \lambda x. \text{if } x \leq k \text{ then } c_1 x \text{ else } (c_2 (m + x - k)) \)
shows trace c \_ 0 (k + l)

proof –
from trace1
interpret trace1: program-trace program-step c \_ 0 k .
from trace2
interpret trace2: program-trace program-step c \_ m l .
{
fix x
assume \( x \)-bound: \( x < (k + 1) \)
have \( c \triangleright_d c \ (\text{Suc} \ x) \)
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: \( k \leq x \)
    by auto
  with \( x \)-bound have bound: \( x - k < 1 \)
    by auto
  from k-x have eq: \( (\text{Suc} \ (m + x) - k) = \text{Suc} \ (m + x - k) \)
    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

\textbf{theorem} \textbf{(in program-progress)} \textbf{safe-free-flowing-implies-safe-delayed:}
\textbf{assumes} init: initial \( c \_0 \)
\textbf{assumes} dist: simple-ownership-distinct (fst \( c \_0 \))
\textbf{assumes} read-only-unowned: read-only-unowned (snd (snd \( c \_0 \))) (fst \( c \_0 \))
\textbf{assumes} unowned-shared: unowned-shared (snd (snd \( c \_0 \))) (fst \( c \_0 \))
\textbf{assumes} safe-reach-ff: safe-reach-upto \( n \) safe-free-flowing \( c \_0 \)
\textbf{shows} safe-reach-upto \( n \) safe-delayed \( c \_0 \)
\textbf{using} safe-reach-ff
\textbf{proof} (induct \( n \))
  case 0
  hence safe-reach-upto 0 safe-free-flowing \( c \_0 \) by simp
  hence safe-free-flowing \( c \_0 \)
    by (auto simp add: program-safe-reach-upto-def)
  from initial-safe-free-flowing-implies-safe-delayed [OF init this]
  have safe-delayed \( c \_0 \).
  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 \( c \_0 \) by simp
  then interpret safe-reach-suc-inter: program-safe-reach-upto program-step (Suc \( n \)) safe-free-flowing \( c \_0 \).
  from safe-reach-upto-le [OF safe-reach-suc]
  have safe-reach-n: safe-reach-upto \( n \) safe-free-flowing \( c \_0 \) by simp
  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-uppto program-step n
safe-delayed c₀.

interpret direct-computation:
computation direct-memop-step empty-storebuffer-step program-step λp p′ is sb. sb.

show ?case
proof (cases safe-reach-uppto (Suc n) safe-delayed c₀)
  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 = c₀ and
  safe-delayed-upto-n: ∀k≤n. safe-delayed (c k) and
  violation-delayed-suc: ¬ safe-delayed (c (Suc n))

proof –
from False
obtain c 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 = c₀
  by (clarsimp simp add: program-safe-reach-uppto-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

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 sb DOR where
  i-bound: $i < \text{length ts}$ and
  ts-i: $ts \mid i = (p, is, \emptyset, sb, D, O, R)$ and
  violation-i: $\neg \text{map owned ts, map released ts } i \vdash \text{(is, } \emptyset, m, D, O, S) \sqrt{\cdot}$
  by (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: $is = \text{ins#is’}$
  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: $\text{map owned ts, i} \vdash (\text{is, } \emptyset, m, D, O, S) \sqrt{\cdot}$.

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

{ assume no-race:
∀ j. j < length ts → j ≠ i → ¬ races (released (ts[j]))
from safe-i
have map owned ts, map released ts, i ⊢ (is, ∅, m, D, O, S)√
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
    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 - 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 < \text{length } \mathit{ts} \text{ and } 
\text{neq-j-i: } j \neq i \text{ and } 
\text{race: } \text{races } \text{(released } ts!j) 
by auto

obtain p_j is_j \theta_j sb_j D_j O_j R_j \text{ where } 
\text{ts-j: } ts!j = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) 
apply \text{(cases } ts!j) 
apply force 
done

from race 
have \mathcal{R}_j\text{-non-empty: } \mathcal{R}_j \neq \text{Map.empty} 
by \text{(auto simp add: ts-j races-def split: instr.splits if-split-asn)}

\{ 
amsume \text{idle-j: } \forall l \leq \text{Suc } n. \ \text{fst } (c l) ! j = \text{fst } (c (\text{Suc } n)) ! j 
have ?thesis 
proof - 
from \text{idle-j [rule-format, of 0] c-suc c-0 ts-j} 
have c_0-j: \text{fst } c_0 ! j = ts!j 
  by clarsimp 
from \text{trace-preserves-length-ts [OF trace, of 0 Suc n] c-0 c-suc} 
have \text{length } (\text{fst } c_0) = \text{length } ts 
  by clarsimp 
with j-bound have j < \text{length } (\text{fst } c_0) 
  by simp 
with nth-mem [OF this] init c_0-j ts-j 
have \mathcal{R}_j = \text{Map.empty} 
  by \text{(auto simp add: initial-def)} 
with \mathcal{R}_j\text{-non-empty have False} 
  by simp 
thus ?thesis ..
qed
\}
moreover 
\{ 
fix last 
assume \text{last-bound: } \text{last}<\text{Suc } n 
assume \text{last-step-changed-j: } \text{fst } (c \text{ last}) ! j \neq \text{fst } (c (\text{Suc } \text{last})) ! j 
assume \text{idle-rest: } \forall l>\text{last. } l \leq \text{Suc } n \rightarrow \text{fst } (c l) ! j = \text{fst } (c (\text{Suc } n)) ! j 
have ?thesis 
proof - 
obtain ts_l S_l m_l \text{ where } 
c-last: c \text{ last } = (ts_l, m_l, S_l) 
  by \text{(cases } c \text{ last)
obtain \( ts'_l \mathcal{S}'_l \mathcal{m}' \) where
\[ c\text{-last}': c \ (\operatorname{Suc\ last}) = (ts'_l,\mathcal{m}'_l,\mathcal{S}'_l) \]
by (cases \( c \ (\operatorname{Suc\ last}) \))
from idle-rest [rule-format, of Suc last ] c-suc c-last' last-bound
have \( ts'_l\cdot j: ts'_l[j] = ts!j \)
by auto

from last-step-changed-j c-last c-last'
have \( j\text{-changed}: ts'_l[j] \neq ts!j \)
by auto

from trace-inter.step [OF last-bound] c-last c-last'
have last-step: \( (ts'_l,\mathcal{m}_l,\mathcal{S}_l) \Rightarrow_d (ts'_l',\mathcal{m}'_l,\mathcal{S}'_l) \)
by simp
obtain \( p_l\ is_l\ \theta_l\ sb_l\ \mathcal{D}_l\ \mathcal{O}_l\ \mathcal{R}_l \) where
\( ts_l\cdot j: ts!_l[j] = (p_l,\is_l,\theta_l,\sb_l,\mathcal{D}_l,\mathcal{O}_l,\mathcal{R}_l) \)
apply (cases \( ts'_l[j] \))
apply force
done

from trace-preserves-length-ts [OF trace, of last Suc n] c-last c-suc last-bound
have leq: \( \text{length } ts_l = \text{length } ts \)
by simp
with \( j\text{-bound} \)

from trace have trace-n: trace \( c \ 0 \ 0 \)
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 \( (ts'_l,\mathcal{m}_l,\mathcal{S}_l) \)
by auto

from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n c-0]
have safe-delayed-upto-n: \( \forall x < n. \text{safe-delayed } (c \ (0 + x)) \)
by auto
from trace-preserves-simple-ownership-distinct [where c=c and n=0 and l=last,
simplified c-0, OF dist trace-n safe-delayed-upto-n]
last-bound c-last
have dist-last: simple-ownership-distinct ts_l
by auto

from trace-preserves-read-only-unowned [where c=c and n=0 and l=last,
simplified c-0, OF dist read-only-unowned trace-n safe-delayed-upto-n]
last-bound c-last
have ro-last-last: read-only-unowned \( \mathcal{S}_l \) ts_l
by auto
from safe-delayed-reach-inter.safe-config [where c=c, OF - trace-n c-0]
have safe-delayed-upto-suc-n: ∀ x < Suc n. 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-n]
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':
  ∀ l ≤ n - last.
    ∀ ts_l S_l m_l. c (Suc last + l) = (ts_l, m_l, S_l) → 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 safe: map owned ts_l,j map released ts_l,j-\j (is_l, \j, 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: ∀ x < n - last. safe-delayed (c (Suc (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_j', \j, \j, 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: \j \vdash p_l →_p (p', is')
  by (cases i' = j) auto
from ts_l' ts_l' ts_l-j j-bound
obtain eqs: p' = p_j is_l @ is_j' = is_j \j = \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 ts-j [simplified], simplified m' S', simplified, OF dist-last dist-last' [simplified ts'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
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. ∀ts_x S_x m_x. c (Suc (last + x)) = (ts_x, m_x, S_x) →
(∀ts_x' S_x' m_x'. c' (Suc (last + x)) = (ts_x', m_x', S_x') →
ts_x' ! j = ts_l ! j ∧
(∀a∈O_l. S_x' a = S_l a) ∧
(∀a∈O_l. m_x a = m_l a) ∧ (∀a∈O_l. m_x a = m_l a))) and

c'-sim:

(∀x≤k. ∀ts_x S_x m_x. c (Suc (last + x)) = (ts_x, m_x, S_x) →
(∀ts_x' S_x' m_x'. c' (Suc (last + x)) = (ts_x', m_x', S_x') →
(∀ja<length ts_x. ja ≠ j → ts_x' ! ja = ts_x ! ja) ∧
(∀a. a /∈ O_l → S_x' a = S_x a) ∧
(∀a. a /∈ O_l → m_x' a = m_x a)))

by auto

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 = 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

95
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, OF c-suc c'-suc]
  obtain u-ts-j: u-ts!j = ts!j and
    shared-UNCH: ∀a∈O. u-shared a = S a and
    shared-orig-UNCH: ∀a∈O. S a = S a and
    mem-UNCH: ∀a∈O. u-m a = m a and
    mem-UNCH-orig: ∀a∈O. m a = m a
      by auto
  from c'-SIM [rule-format, OF k, simplified k, OF c-suc c'-suc]
  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: \( \forall x \leq n. \) safe-delayed \( (c\text{-undo } x) \)
by (auto simp add: c-undo split: if-split-asm)
hence safe-delayed-c-undo: \( \forall x < n. \) safe-delayed \( (c\text{-undo } x) \)
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
} hence u-shared-eq: u-shared = \( S \) by -(rule ext, auto)
{ 
assume safe: map owned u-ts, map released u-ts, i ⊢ (is, 0, 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
\begin{verbatim}
}\)
\begin{proof}
\end{proof}
\end{verbatim}
obtain $c' \ k$ where
k-bound: $k \leq n - \text{last}$ and
trace-$c'$: trace $c' (\text{Suc last}) \ k$ and
c'-first: $c' (\text{Suc last}) = (ts_1, m_1, S_1)$ and
c'-leq: $(\forall x \leq k. \ \text{length (fst (c' (Suc (last + x))))} = \text{length (fst (c (Suc (last + x))))})$ and
c'-safe: $(\forall x < k. \ \text{safe-delayed (c' (Suc (last + x))))}$ and
c'-unsafe: $(k < n - \text{last} \rightarrow \neg \text{safe-delayed (c' (Suc (last + k))))}$ and
c'-unch:
$(\forall x \leq k. \ \forall ts_x S_x m_x. \ c (\text{Suc (last + x)}) = (ts_x, m_x, S_x) \rightarrow$
$(\forall ts_x' S_x' m_x'. \ c' (\text{Suc (last + x)}) = (ts_x', m_x', S_x') \rightarrow$
$ts_x' ! j = ts_1 ! j \land$
$(\forall a \in O_1. \ S_x' a = S_1 a) \land$
$(\forall a \in O_1. \ S_x a = S_1 a) \land$
$(\forall a \in O_1. \ m_x' a = m_1 a) \land (\forall a \in O_1. \ m_x a = m_1 a)))$ and
c'-sim:
$(\forall x \leq k. \ \forall ts_x S_x m_x. \ c (\text{Suc (last + x)}) = (ts_x, m_x, S_x) \rightarrow$
$(\forall ts_x' S_x' m_x'. \ c' (\text{Suc (last + x)}) = (ts_x', m_x', S_x') \rightarrow$
$(\forall j a < \text{length ts}_x. \ j a \neq j \rightarrow ts_x' ! j a = ts_x ! j a) \land$
$(\forall a. \ a \notin O_1 \rightarrow S_x' a = S_x a) \land$
$(\forall a. \ a \notin O_1 \rightarrow m_x' a = m_x a)))$
by (clarsimp simp add: $O_1'$)

obtain c-undo where c-undo: $c$-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-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: $\neg$ safe-delayed (c-undo (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
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 O_1. u-shared a = S a and
    shared-orig-unch: \forall a \in O_1. S a = S one a and
    mem-unch: \forall a \in O_1. u-m a = m a and
    mem-unch-orig: \forall a \in O_1. m a = m one 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. a /\in O_1 \rightarrow u-shared a = S a and
    mem-sim: \forall a. a /\in O_1 \rightarrow 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  
  have u-m a = m a  
  proof (cases a ∈ \(\mathcal{O}_l\))  
    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 = \(\mathcal{S}\) a  
  proof (cases a ∈ \(\mathcal{O}_l\))  
    case True with shared-unch  
    have u-shared a = \(\mathcal{S}_l\) a  
    by auto  
    moreover  
    from True shared-orig-unch  
    have \(\mathcal{S}\) a = \(\mathcal{S}_l\) 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 = \(\mathcal{S}\) by – (rule ext, auto)

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

hence \(\neg\) \(\text{safe-delayed (u-ts, u-m, u-shared)}\)
apply (clarsimp simp add: \(\text{safe-delayed-def}\))
apply (rule-tac \(x=i\) \(\text{in exI}\))
using \(u\text{-}ts\text{-}i \ ts\text{-}i \ i\text{-}bound\text{-}u\)
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 (WriteNonVolatile a D f A L R W)
then obtain
  is': is' = Write False a (D, f) A L R W # is' and
  A': A' = A and
  m': m' = m(a := f θ)
  by auto
note eqs' = θ A' D' O' R' S' m'
from ts'-j ts'-j j-bound' eqs'
obtain eqs': p1=pj is'=is' θA=θj D'j=Dj O'j=O'j
R'=Rj
  by auto

from safe1 [simplified is] obtain a-owned: a ∈ O' and a-unshared: a /∈ dom S'
  by cases auto
have m'-unch-unowned: ∀ a'. a' /∈ O' → m[a'] = (m(a := f θ)) a'
  using a-owned by auto
have m'-unch-unshared: ∀ a'. a' ∈ O' → a' ∈ dom S' → m[a'] = (m(a := f θ)) 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'],
simplified,
  OF j-bound ts'-j [simplified], simplified m' S',OF m'-unch-unowned m'-unch-unshared, simplified,
  OF dist-last dist-last' [simplified ts',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', m', S') 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: \((k < n - \text{last} \rightarrow \neg \text{safe-delayed} \ (c' \ (\text{Suc} \ (\text{last} + k))))\) and \\
\(c'\)-unch: \\
\((\forall x \leq k. \ \forall t_s \ S_x \ m_x. \ c (\text{Suc} \ (\text{last} + x)) = (t_s, \ m_x, \ S_x) \rightarrow \ c' (\text{Suc} \ (\text{last} + x)) = (t_s', \ m_x', \ S_x') \rightarrow \) \\
\(\neg \text{safe-delayed} \ (c' (\text{Suc} \ (\text{last} + k)))\) and \\
\(c'\)-unch: \\
\((\forall x \leq k. \ \forall t_s \ S_x \ m_x. \ c (\text{Suc} \ (\text{last} + x)) = (t_s, \ m_x, \ S_x) \rightarrow \ c (\text{Suc} \ (\text{last} + x)) = (t_s', \ m_x', \ S_x') \rightarrow \) \\
\(\forall a \in O, \ S_x' a = S_l a \) \land \\
\(\forall a \in O, \ S_x a = S_l a \) \land \\
\(\forall a \in O, \ m_x' a = m_l a \) \land \ (\forall a' \in O, \ m_x a' = (m_l (a := f \theta_l)) a'))\) and \\
\(c'\)-sim: \\
\((\forall x \leq k. \ \forall t_s \ S_x \ m_x. \ c (\text{Suc} \ (\text{last} + x)) = (t_s, \ m_x, \ S_x) \rightarrow \ c (\text{Suc} \ (\text{last} + x)) = (t_s, \ m_x, \ S_x) \rightarrow \) \\
\(\forall a \in O, \ S_x' a = S_x a \) \land \ (\forall a \in O, \ m_x' a = m_x a))\) \\
by \(\text{clarsimp simp add: } O_l'\) \\
\begin{align*}
\text{obtain } & c\text{-undo where } 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})) \\
& \text{by } \text{blast} \\
& \text{have } c\text{-undo-0: } c\text{-undo } 0 = c_0 \\
& \quad \text{by } (\text{auto simp add: } c\text{-undo c-0}) \\
& \text{from } \text{sequence-traces } [\text{OF trace-last trace-c', simplified c-last, OF c'-first c-undo}] \\
& \text{have } \text{trace-undo: trace } c\text{-undo } 0 \ (\text{last} + k) \ . \\
& \text{obtain } u-t_s \ u\text{-shared } u\text{-m where} \\
& \quad c\text{-undo-n: } c\text{-undo } n = (u-t_s, u-m, u\text{-shared}) \\
& \quad \text{by } (\text{cases c-undo } n) \\
& \text{with } \text{last-bound c'-first c-last} \\
& \text{have } c\text{-suc: } c' (\text{Suc } n) = (u-t_s, u-m, u\text{-shared}) \\
& \quad \text{apply } (\text{auto simp add: } c\text{-undo split: if-split-asm}) \\
& \quad \text{apply } (\text{subgoal-tac n=last}) \\
& \quad \text{apply } \text{auto} \\
& \text{done} \\
\end{align*} \\
\begin{align*}
\text{show } & \ ?\text{thesis} \\
\text{proof } & (\text{cases } k < n - \text{last}) \\
& \text{case True} \\
& \text{with } c'\text{-unsafe have unsafe: } \neg \text{safe-delayed} \ (c\text{-undo } (\text{last} + k)) \\
& \quad \text{by } (\text{auto simp add: } c\text{-undo c-last c'-first}) \\
& \text{from } \text{True have } \text{last + k } \leq n \\
& \quad \text{by } \text{auto} \\
& \text{from } \text{safe-delayed-reach-inter.safe-config } [\text{OF this trace-undo, of last + k}] \\
& \text{have } \text{safe-delayed} \ (c\text{-undo } (\text{last} + k)) \\
& \quad \text{by } (\text{auto simp add: } c\text{-undo c-0}) \\
& \text{with } \text{unsafe have False by simp} \\
& \text{thus } ?\text{thesis } .. \\
\end{align*}
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: \(\forall a\in\mathcal{O}_l\). u-shared a = \mathcal{S}_l a **and**

shared-orig-unch: \(\forall a\in\mathcal{O}_l\). \mathcal{S} a = \mathcal{S}_l a **and**

mem-unch: \(\forall a\in\mathcal{O}_l\). u-m a = m_l a **and**

mem-unch-orig: \(\forall a'\in\mathcal{O}_l\). m a' = (m_l(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. a /\in \mathcal{O}_l \rightarrow u-shared a = \mathcal{S} a **and**

mem-sim: \(\forall a. a /\in \mathcal{O}_l \rightarrow 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

**have** u-shared a = \mathcal{S} a

**proof** (cases a \in \mathcal{O}_l)

**case** True **with** shared-unch

106
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 ?thesis

  by auto

qed

} hence \( u\text{-}shared\text{-eq: } u\text{-}shared = \mathcal{S} \ by \ (\text{rule ext, auto) }

{ 

assume safe: map owned u-ts, map released u-ts, i \vdash (\text{is, } \theta, u\text{-}m, \mathcal{D}, \mathcal{O}, u\text{-}shared) \checkmark

then have False

proof cases

  case Read

  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)

next

case WriteNonVolatile

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)

next

case WriteVolatile

then show ?thesis

using \( \text{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \)

apply (auto simp add:eqs races-def split: if-split-asm)

apply fastforce

done

next

case Fence

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)

next

case Ghost

then show ?thesis

using \( \text{ts-i ts-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \)

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)

with ts-i is obtain
ins: \( \text{ins} = \text{RMW} \ a' \ t \ (D, \ f) \ \text{cond ret A L R W} \) and

owned-or-shared: \( a' \in \mathcal{O} \lor a' \in \text{dom u-shared} \) and

cond: \( \neg \text{cond (} \theta(t \mapsto u-m \ a') \) and

rels-race: \( \forall j < \text{length (map owned u-ts). } i \neq j \rightarrow ((\text{map released u-ts}) ! j) \)

\( a' \neq \text{Some False} \)

by auto

from \( \text{dist-u-ts-inter.simple-ownership-distinct [OF j-bound-u i-bound-u neq-j-i u-ts-j [simplified ts-i-j]}

\( \text{u-ts-i [simplified ts-i]]} \)

have \( \text{dist: } \mathcal{O}_I \cap \mathcal{O} = \{\} \)

by auto

from owned-or-shared dist a-owned a-unshared shared-orig-unch

have \( a' \neq a \)

by (auto simp add: u-shared-eq domIff)

have \( u-m \ a' = m \ a' \)

proof (cases \( a' \in \mathcal{O}_I \))

case True with mem-unch

have \( u-m \ a' = m \ a' \)

by auto

moreover

from True mem-unch-orig a'-a

have \( m \ a' = m \ a' \)

by auto

ultimately show \( ?\text{thesis by simp} \)

next

case False

with mem-sim

show \( ?\text{thesis} \)

by auto

qed

with \( \text{ins cond rels-race show } ?\text{thesis} \)

using \( \text{ts-i ts-s-j race is j-bound i-bound u-ts-i u-ts-j leq-u-ts neq-j-i ts-j} \)

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 \( \text{ts-i is obtain} \)

ins: \( \text{ins} = \text{RMW} \ a' \ t \ (D, \ f) \ \text{cond ret A L R W and} \)

\( \text{cond: } \neg \text{cond (} \theta(t \mapsto u-m \ a') \) and

\( a' \forall j < \text{length (map owned u-ts). } i \neq j \rightarrow a' \notin (\text{map owned u-ts}) ! j \cup \text{dom} \)

\( ((\text{map released u-ts}) ! j) \) and

safety:

\( A \subseteq \text{dom u-shared} \cup \mathcal{O} \cup L \subseteq A \ R \subseteq \mathcal{O} \ A \cap R = \{\} \)

\( \forall j < \text{length (map owned u-ts). } i \neq j \rightarrow A \cap ((\text{map owned u-ts}) ! j \cup \text{dom} \)

\( ((\text{map released u-ts}) ! j) = \{\} \)

\( a' \notin \text{read-only u-shared} \)

by auto

from \( \text{a [rule-format, of j] j-bound-u u-ts-j ts-s-j neq-j-i} \)

have \( a' \notin \mathcal{O}_I \)

by auto

from \( \text{mem-sim [rule-format, OF this]} \)
have u-m-eq: u-m a' = m a'
by auto

with ins cond safety a' show \textit{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-asym)
apply fastforce
done
next
case Nil
then show \textit{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-asym)
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 \textit{thesis}
by simp
qed
next
case WriteVolatile
with ts' j ts' ts-j j-bound_i have R_j = Map.empty
by auto
with R_i-non-empty have False by auto
thus \textit{thesis} ..
next
case Fence
with ts' j ts' ts-j j-bound_i have R_j = Map.empty
by auto
with R_i-non-empty have False by auto
thus \textit{thesis} ..
next
case RMWReadOnly
with ts' j ts' ts-j j-bound_i have R_j = Map.empty
by auto
with R_i-non-empty have False by auto
thus \textit{thesis} ..
next
case RMWWrite
with \( \text{ts}_l \cdots j \text{ts}_l \) have \( \mathcal{R}_j = \text{Map.empty} \)
by auto
with \( \mathcal{R}_j \)-non-empty have False by auto
thus thesis ..

next
case (Ghost A L R W)
then obtain
is!: is! = \( \text{Ghost A L R W} \) # is!' and
\( \partial_l', \partial_j' = \partial_l \) and
sb!: sb!' = sb! and
\( D_l' = D_l \) and
\( O_l' = O_l \cup A - R \) and
\( R_l' = \text{augment-rels} (\text{dom } S_l) R R_l \) and
\( S_l' = S_l \oplus W R \ominus A L \) and
m!: m!' = m!
by auto
note eqs' = \( \partial_l' \) sb!' D_l' O_l' R_l' S_l' m'
from ts'-j ts' j-bound_j eqs'
obtain eqs: \( p_j = p_j \) is! = is! \( \partial_l = \partial_j \) D_l = D_j O_l \cup A - R = O_j
augment-rels (\text{dom } S_l) R R_l = R_l by auto

from safe_l [simplified is!]
obtain
A-shared-owned: \( A \subseteq \text{dom } S_l \cup O_l \) and L-A: \( L \subseteq A \) and R-owns: \( R \subseteq O_l \) and
A-R: \( A \cap R = \{\} \) and
\( \forall j' < \text{length} (\text{map owned } \text{ts}_l). j \neq j' \rightarrow A \cap ((\text{map owned } \text{ts}_l)!j' \cup \text{dom } ((\text{map released } \text{ts}_l)!j')) = \{\} \)
by cases auto

from undo-local-steps \[\text{where } c = c, \text{OF trace-rest } c \text{-last'} idle-rest' safe-delayed-last, simplified } \text{ts}_l'\],
simplified,
\( \text{OF } j\text{-bound}_l \text{ts}_l \text{-j}[\text{simplified}], \text{simplified } m!' S_l', \text{simplified,} \)
\( \text{OF } \text{shared-eq dist-last dist-last'} [\text{simplified ts}_l'[\text{simplified]} \text{safe-delayed-upto-last}] \)

obtain \( c' \) k where
k-bound: \( k \leq n - \text{last} \) and
trace-c': trace \( c' (\text{Suc last}) \) k and
c'-first: \( c' (\text{Suc last}) = (\text{ts}_l, m!, S_l) \) and
c'-leq: \( (\forall x \leq k, \text{length } (\text{fst } (c' (\text{Suc } (\text{last} + x)))) = \text{length } (\text{fst } (c (\text{Suc } (\text{last} + x)))) \)) and
\( c'\text{-safe: } (\forall x < k, \text{safe-delayed } (c' (\text{Suc } (\text{last} + x)))) \) and
c'-unsafe: \( (k < n - \text{last } 
\rightarrow \neg \text{safe-delayed } (c' (\text{Suc } (\text{last} + k)))) \) and
c'-unch:
(∀ x ≤ k. ∀ ts x S x m x.
c (Suc (last + x)) = (ts x, m x, S x) →
(∀ ts x' S x' m x'.
c' (Suc (last + x)) = (ts x', m x', S x') →
ts x' ! j = ts l ! j ∧
(∀ a ∈ O l, S x' a = S l a) ∧
(∀ a ∈ O l, S x a = (S l ⊕ W R ⊕ A l) a) ∧
(∀ a ∈ O l, m x' a = m l a) ∧
(∀ a ∈ O l, m x a = (m l) a)) and
by (clarsimp )
obtain c-undo where
c-undo: c-undo = (λ x. if x ≤ 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

111
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 O_l. u-shared a = S_l a and
  shared-orig-unch: \forall a \in O_l. \mathcal{S} a = (S_l \oplus W R \ominus A L) a and
  mem-unch: \forall a \in O_l. u-m a = m_l a and
  mem-unch-orig: \forall a' \in O_l. m a' = m_l a'
  by auto
from c' - sim [rule-format, of k, simplified k eq', OF - c-suc c' - suc]
obtain u-ts-i: u-ts!i = ts!i [and]
  shared-sim: \forall a. a/ \in O_l \longrightarrow \in O_l' \longrightarrow u-shared a = S a and
  mem-sim: \forall a. a/ \notin O_l \longrightarrow 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
  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 t 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 ⊦(is,θ,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 ∈ O ∨ a ∈ read-only u-shared ∨ volatile ∧ a ∈ dom u-shared
    and
    rels-cond: ∀ j<br>length u-ts. i ≠ j → ((map released u-ts) ! j) a ≠ Some False
    and
    rels-non-volatile-cond: ¬ volatile → (∀ j<br>length u-ts. i ≠ j → a /∈ dom ((map released u-ts) ! j) )
  and
    clean: volatile → ¬ D
  by auto
from race ts-j
have rc: augmen-rels (dom S i) R t a = Some False ∨
  (¬ volatile ∧ a ∈ dom (augmen-rels (dom S i) R t 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-r-j j-bound-u
have R t-a: R t a ≠ Some False
  by auto
u-ts-i [simplified ts-i]
have dist: O i ∩ O = {}
  by auto
show ?thesis
proof (cases volatile)
  case True
  note volatile=this
  show ?thesis
  proof (cases a ∈ R)
    case False
    with rc R t-a show False
      by (auto simp add: augmen-rels-def volatile)
next
  case True
  with R-owns
  have a-owns\_l: a ∈ 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 ∉ O
    by auto
moreover
{
  assume a ∈ 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\_l-a have False
    by (auto simp add: augment-rels-def split: option.splits simp add: domIff volatile)
}
moreover
{
  assume a ∈ dom u-shared
  with u-shared-eq rc True R\_l-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\_j j-bound-u
  have R\_l-a: R\_l a = None
    by (auto simp add: domIff)
  show ?thesis
proof (cases a ∈ R)
case False
  with rc R\_l-a show False
    by (auto simp add: augment-rels-def non-volatile domIff)
next
case True
  with R-owns
  have a-owns\_l: a ∈ 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

114
from a-owns \_i dist have a-unowned: a \notin O 
  by auto
moreover
from ro-last-last interpret 
read-only-unowned S \_i ts \_i .
  from read-only-unowned \[OF j-bound \_i ts \_i-j\] a-owns \_i have a-unsh: a \notin 
read-only S \_i by auto
  
\{
  assume a \in read-only u-shared 
  with u-shared-eq have sh: S \_i a = Some False
  by (auto simp add: read-only-def)
  
  with rc True \(\mathcal{R}_i\)-a access-cond u-shared-eq a-unowned sh a-owns \_i 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 \(\mathcal{R}_i\)-a a-owns \_i 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-i]
  have dist: O \_i \cap O = \{
  by auto
from race ts-j 
  have rc: a \in dom (augment-rels (dom S \_i) R \_i)
  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 \(\mathcal{R}_i\)-a: a \notin dom \(\mathcal{R}_i\)
  by auto
show False

115
proof (cases a \in R)
case False
with rc \mathcal{R}_l-a show False
  by (auto simp add: augment-rels-def domIff)
next
case True
with R-owns
have a-owns\_l: a \in \mathcal{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) \land a \notin \text{dom } ((\text{map released } u\_ts)! j) and
A'\text{-owns-shared}: A' \subseteq \text{dom } u\_shared \cup \mathcal{O} \ and
L'\text{-A'}: L' \subseteq A' \ and
R'\text{-owned}: R' \subseteq \mathcal{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 } ((\text{map released } u\_ts)! j)) = \{} and
writeable: a \notin \text{read-only } u\_shared
  by auto
from a-un-owned-released \ [\text{rule-format, simplified, OF j-bound-u neq-j-i [symmetric]}] u\_ts-j ts\_i-j j-bound-u
obtain \mathcal{O}_l-a: a \notin \mathcal{O}_l \ and \ \mathcal{R}_l-a: a \notin \text{dom } (\mathcal{R}_l)
  by auto
from acq-ok \ [\text{rule-format, simplified, OF j-bound-u neq-j-i [symmetric]}] u\_ts-j ts\_i-j j-bound-u
obtain \mathcal{O}_l-A': A' \cap \mathcal{O}_l = \{} and \ \mathcal{R}_l-A': A' \cap \text{dom } (\mathcal{R}_l) = \{}
  by auto
{
  assume rc: a \in \text{dom } (\text{augment-rels } (\text{dom } \mathcal{S}_l) \ R \ \mathcal{R}_l)
  have False
proof (cases a \in R)
case False
  with rc \mathcal{R}_l-a show False
    by (auto simp add: augment-rels-def domIff)
next
case True
  with R-owns
  have a-owns\_l: a \in \mathcal{O}_l
    by auto
  with \mathcal{O}_l-a show False
    by auto
qed

116
moreover
{
  assume rc: \( A' \cap \text{dom} (\text{augment-rels} (\text{dom} S_l) R R_l) \neq \{\} \)
  then obtain a' where a'-A': a' \in A' and a'-aug: a' \in \text{dom} (\text{augment-rels} (\text{dom} S_l) R R_l)
  by auto
  have False
  proof (cases a' \in R)
    case False
    with a'-aug a'-A' \( \cap R_l \) show False
    by (auto simp add: augment-rels-def domIff)
  next
    case True
    with R-owns have a'-owns_l: a' \in \mathcal{O}_l
    by auto
    with \mathcal{O}_l 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 \mathcal{O} and
L'-A': L' \subseteq A' and
R'-owned: R' \subseteq \mathcal{O} and
A'-R': A' \cap R' = \{\} and
acq-ok: \( \forall j < \text{length} u-ts. \ i \neq j \rightarrow A' \cap ((\text{map} \text{owned} \text{u-ts})!j \cup \text{dom} ((\text{map} \text{released} \text{u-ts})!j)) = \{\} \)
by auto
from acq-ok [rule-format, simplified, OF j-bound-u neq-j-i [symmetric]] u-ts-j ts-l-j j-bound-u
obtain \mathcal{O}_l-A': A' \cap \mathcal{O}_l = \{\} and \( \mathcal{R}_l-A': A' \cap \text{dom} (\mathcal{R}_l) = \{\}
by auto
from race ts-j
obtain a' where a'-A': a' \in A' and
a'-aug: a' \in \text{dom} (\text{augment-rels} (\text{dom} S_l) R R_l)
by (auto simp add: races-def ins eqs)
show False
proof (cases a' \in R)
case False
  with a′-aug a′-A′ R_l-A′ show False
    by (auto simp add: augment-rels-def domIff)
next
case True
  with R-owns have a′-owns: a′ ∈ O_l
    by auto
  with O_l-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-l-j]]
      u-ts-i [simplified ts-i]]
    have dist: O_l ∩ O = {}
      by auto
    from race ts-j cond
    have rc: augment-rels (dom S_l) 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_l-j 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 ∈ O_l
    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
    by auto
  with u-shared-eq re True R_l-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 (θ(t ↦→ u-m a)) and
a-un-owned-released: ∀j<length (map owned u-ts). i ≠ j → a /∈ (map owned u-ts) ! j ∪ dom ((map released u-ts) ! j) and
A'-owns-shared: A' ⊆ dom u-shared ∪ O and
L'-A': L' ⊆ A' and
R'-owned: R' ⊆ O and
A'-R': A' ∩ R' = {} and
acq-ok: ∀j<length (map owned u-ts). i ≠ j → A' ∩ ((map owned u-ts) ! j ∪ dom ((map released u-ts) ! j)) = {} and
writeable: a /∈ 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 /∈ O_l and R_l-a: a /∈ 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' ∩ O_l = {} and R_l-A': A' ∩ dom (R_l) = {}
by auto
{
assume rc: a ∈ dom (augment-rels (dom S_l) R R_l)
have False
proof (cases a ∈ R)
case False
with rc R_l-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 show False
by auto
qed
}
moreover
{
assume rc: A’ ∩ dom (augment-rels (dom S_l) R R_l) ≠ {}
then obtain a’ where a'-A': a’ ∈ A’ and a'-aug: a’ ∈ dom (augment-rels (dom S_l) R R_l)
by auto
have False
proof (cases a’ ∈ R)
case False
  with a'-aug a'-A' R-i-A' show False
  by (auto simp add: augment-rels-def domIff)
next
  case True
    with R-owns have a'-owns: a' ∈ O_i
    by auto
    with O_i-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
}
ext
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
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
(- → f - [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)

abbreviation flush-steps::memory × 'p store-buffer × owns × rels × shared ⇒ memory × 'p store-buffer × owns × rels × shared⇒ bool
(- → f* - [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^**

term x → w* Y
fun buffered-val :: 'p memref list ⇒ addr ⇒ val option
where
  buffered-val [] a = None
| buffered-val (r # rs) a' =
  (case r of
    Write sb volatile a - v - - - - ⇒ (case buffered-val rs a' of
      None ⇒ (if a' = a then Some v else None)
      | Some v' ⇒ Some v'))
| - ⇒ buffered-val rs a')

definition address-of :: 'p memref ⇒ addr set
where
  address-of r = (case r of
    Write sb volatile a - v - - - - ⇒ {a}
    | Read sb volatile a t v ⇒ {a}
    | - ⇒ {}})

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

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

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

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

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

definition is-Prog sb :: 'p memref ⇒ bool
where
  is-Prog sb r = (case r of
    Prog sb - - - - ⇒ True
    | - ⇒ False)

definition is-Ghost sb :: 'p memref ⇒ bool
where
  is-Ghost sb r = (case r of
    Ghost sb - - - - ⇒ True
    | - ⇒ False)
lemma is-Write_{sb}-simps [simp]:
is-Write_{sb} (Write_{sb} volatile a sop v A L R W) = True
is-Write_{sb} (Read_{sb} volatile a t v) = False
is-Write_{sb} (Prog_{sb} p p' is) = False
is-Write_{sb} (Ghost_{sb} A L R W) = False
  by (auto simp add: is-Write_{sb}-def)

lemma is-Read_{sb}-simps [simp]:
is-Read_{sb} (Read_{sb} volatile a t v) = True
is-Read_{sb} (Write_{sb} volatile a sop v A L R W) = False
is-Read_{sb} (Prog_{sb} p p' is) = False
is-Read_{sb} (Ghost_{sb} A L R W) = False
  by (auto simp add: is-Read_{sb}-def)

lemma is-Prog_{sb}-simps [simp]:
is-Prog_{sb} (Read_{sb} volatile a t v) = False
is-Prog_{sb} (Write_{sb} volatile a sop v A L R W) = False
is-Prog_{sb} (Prog_{sb} p p' is) = True
is-Prog_{sb} (Ghost_{sb} A L R W) = False
  by (auto simp add: is-Prog_{sb}-def)

lemma is-Ghost_{sb}-simps [simp]:
is-Ghost_{sb} (Read_{sb} volatile a t v) = False
is-Ghost_{sb} (Write_{sb} volatile a sop v A L R W) = False
is-Ghost_{sb} (Prog_{sb} p p' is) = False
is-Ghost_{sb} (Ghost_{sb} A L R W) = True
  by (auto simp add: is-Ghost_{sb}-def)

definition is-volatile-Write_{sb} :: 'p memref ⇒ bool
where
is-volatile-Write_{sb} r = (case r of Write_{sb} volatile a - v - - - -⇒ volatile | - ⇒ False)

lemma is-volatile-Write_{sb}-simps [simp]:
is-volatile-Write_{sb} (Write_{sb} volatile a sop v A L R W) = volatile
is-volatile-Write_{sb} (Read_{sb} volatile a t v) = False
is-volatile-Write_{sb} (Prog_{sb} p p' is) = False
is-volatile-Write_{sb} (Ghost_{sb} A L R W) = False
  by (auto simp add: is-volatile-Write_{sb}-def)

lemma is-volatile-Write_{sb}-address-of [simp]: is-volatile-Write_{sb} x =⇒ address-of x ≠ {}
  by (cases x) auto

definition is-volatile-Read_{sb} :: 'p memref ⇒ bool
where
is-volatile-Read_{sb} r = (case r of Read_{sb} volatile a t v ⇒ volatile | - ⇒ False)

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-append: 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-append)
lemma outstanding-refs-empty-negate: (outstanding-refs P sb = {}) \implies
(outstanding-refs (Not \circ P) sb = \bigcup \{\text{address-of } i \mid \text{set sb}\})
by (auto simp add: outstanding-refs-conv)

lemma outstanding-refs-mono-pred:
\( \forall sb \, sb'. \forall r \mid P \to P' \, r \implies \text{outstanding-refs } P \, sb \subseteq \text{outstanding-refs } P' \, sb' \)
by (auto simp add: outstanding-refs-conv)

lemma outstanding-refs-mono-set:
\( \forall sb \, sb'. \text{set sb} \subseteq \text{set sb}' \implies \text{outstanding-refs } P \, sb \subseteq \text{outstanding-refs } P \, sb' \)
by (auto simp add: outstanding-refs-conv)

lemma outstanding-refs-takeWhile:
\( \text{outstanding-refs } P \, \text{(takeWhile } P' \, sb) \subseteq \text{outstanding-refs } P \, sb \)
apply (rule outstanding-refs-mono-set)
apply (auto dest: set-takeWhileD)
done

lemma outstanding-refs-subsets:
\( \text{outstanding-refs is-volatile-Write}_{sb} \, sb \subseteq \text{outstanding-refs is-Write}_{sb} \, sb \)
\( \text{outstanding-refs is-non-volatile-Write}_{sb} \, sb \subseteq \text{outstanding-refs is-Write}_{sb} \, sb \)
\( \text{outstanding-refs is-volatile-Read}_{sb} \, sb \subseteq \text{outstanding-refs is-Read}_{sb} \, sb \)
\( \text{outstanding-refs is-non-volatile-Read}_{sb} \, sb \subseteq \text{outstanding-refs is-Read}_{sb} \, sb \)
\( \text{outstanding-refs is-volatile-Write}_{sb} \, sb \subseteq \text{outstanding-refs (is-volatile) sb} \)
\( \text{outstanding-refs is-volatile-Read}_{sb} \, sb \subseteq \text{outstanding-refs (is-volatile) sb} \)
\( \text{outstanding-refs is-non-volatile-Write}_{sb} \, sb \subseteq \text{outstanding-refs (Not } \circ \text{is-volatile) sb} \)
\( \text{outstanding-refs is-non-volatile-Read}_{sb} \, sb \subseteq \text{outstanding-refs (Not } \circ \text{is-volatile) sb} \)
by (auto intro!:outstanding-refs-mono-pred simp add: is-volatile-Write_{sb}-def is-non-volatile-Write_{sb}-def is-volatile-Read_{sb}-def is-non-volatile-Read_{sb}-def split: memref.splits)

lemma outstanding-non-volatile-refs-conv:
\( \text{outstanding-refs (Not } \circ \text{is-volatile) sb} = \text{outstanding-refs is-non-volatile-Write}_{sb} \, sb \cup \text{outstanding-refs is-non-volatile-Read}_{sb} \, sb \)
apply (induct sb)
apply simp

125
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 ∪ 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 ∪ 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 ∪ 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 ◦ is-volatile-Read_{sb}) sb =
  outstanding-refs is-Write_{sb} sb ∪ outstanding-refs is-non-volatile-Read_{sb} sb
apply (induct sb)
apply (clarsimp)
subgoal for a sb
  by (case-tac a, auto)
done

lemmas misc-outstanding-refs-convs = outstanding-non-volatile-refs-conv
outstanding-volatile-refs-conv
outstanding-is-Write_{sb}-refs-conv
outstanding-not-volatile-Read_{sb}-refs-conv

lemma no-outstanding-vol-write-takeWhile-append: outstanding-refs is-volatile-Write_{sb} sb = {} ===>
takeWhile (Not ◦ is-volatile-Write\sb) (\sb@\xs) = \sb@(takeWhile (Not ◦ 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 \sb \neq \{\} \implies
takeWhile (Not ◦ is-volatile-Write\sb) (\sb@\xs) = (takeWhile (Not ◦ 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 ◦ is-volatile-Write\sb) (\sb@\xs) = (dropWhile (Not ◦ 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 \sb \neq \{\} \implies
dropWhile (Not ◦ is-volatile-Write\sb) (\sb@\xs) = (dropWhile (Not ◦ 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 ◦ is-volatile-Write\sb) \sb) =
outstanding-refs is-Write\sb (takeWhile (Not ◦ 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\sb \sb = \{\} \implies (dropWhile (Not ◦ is-volatile-Write\sb) \sb) = []
apply (induct sb)
apply (auto split: if-split-asm)
done

lemma takeWhile-not-vol-write-outstanding-refs:
outstanding-refs is-volatile-Write \sb (takeWhile (Not is-volatile-Write \sb) \sb) = {}
apply (induct sb)
apply (auto split: if-split-asm)
done

lemma no-volatile-Write \sb-s-conv: (outstanding-refs is-volatile-Write \sb \sb = {}) =
(\forall r \in \text{set} \ sb. (\forall v' \text{sop}' a' A L R W. r \neq \text{Write}_{\sb} \text{True} a' \text{sop}' v A L R W))
by (force simp add: outstanding-refs-conv is-volatile-Write \sb-def split: memref.splits)

lemma no-volatile-Read \sb-s-conv: (outstanding-refs is-volatile-Read \sb \sb = {}) =
(\forall r \in \text{set} \ sb. (\forall t' a'. r \neq \text{Read}_{\sb} \text{True} a' t' v'))
by (force simp add: outstanding-refs-conv is-volatile-Read \sb-def split: memref.splits)

inductive sb-memop-step :: (instrs × tmps × 'p store-buffer × memory × 'dirty × 'owns × 'rels × 'shared ) ⇒ (instrs × tmps × 'p store-buffer × memory × 'dirty × 'owns × 'rels × 'shared ) ⇒ bool
to sb - [60,60] 100
where
SBReadBuffered:
[buffered-val sb a = Some v]
⇒
(Read volatile a t # is, \emptyset, sb, m, D, O, R, S) →_{sb}
(is, \emptyset (t⇒v), sb, m, D, O, R, S)

| SBReadUnbuffered:
[buffered-val sb a = None]
⇒
(Read volatile a t # is, \emptyset, sb, m, D, O, R, S) →_{sb}
(is, \emptyset (t⇒m a), sb, m, D, O, R, S)

| SBWriteNonVolatile:
(Write False a (D,f) A L R W # is, \emptyset, sb, m, D, O, R, S) →_{sb}
(is, \emptyset, sb@ \text{Write}_{sb} \text{False} a \text{D,f} A L R W, m, D, O, R, S)

| SBWriteVolatile:
(Write True a (D,f) A L R W # is, \emptyset, sb, m, D, O, R, S) →_{sb}
(is, \emptyset, sb@ \text{Write}_{sb} \text{True} a \text{D,f} A L R W, m, D, O, R, S)

| SBFence:
(Fence # is, \emptyset, [], m, D, O, R, S) →_{sb} (is, \emptyset, [], m, D, O, R, S)
SBRMWReadOnly:
\[\neg \text{cond} \ (\theta (t \mapsto m \ a)) \implies \text{RMW} \ a \ t \ (D, f) \text{ cond ret} A \ L \ R \ W \# \text{ is} \ \emptyset, [], m, D, O, R, S) \rightarrow_{sb} \text{is} \ \emptyset, \emptyset, m, D, O, R, S, Sb\]

SBRMWWrite:
\[\text{cond} \ (\theta (t \mapsto m \ a)) \implies \text{RMW} \ a \ t \ (D, f) \text{ cond ret} A \ L \ R \ W \# \text{ is} \ \emptyset, [], m, D, O, R, S) \rightarrow_{sb} \text{is} \ \emptyset, \emptyset, m, D, O, R, S, Sb\]

SBGhost:
\[
(\text{Ghost} \ A \ L \ R \ W \# \text{ is} \ \emptyset, \emptyset, m, D, O, R, S) \rightarrow_{sb} \text{is} \ \emptyset, \emptyset, m, D, O, R, S, Sb\]

\text{inductive} \ s bh\text{-memop-step}::
\[(\text{instrs} \times \text{tmps} \times \ 'p' \text{ store-buffer} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{rels} \times \text{shared}) \Rightarrow \text{instrs} \times \text{tmps} \times \ 'p' \text{ store-buffer} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{rels} \times \text{shared} \Rightarrow \text{bool} \ (\rightarrow_{s bh} - [60, 60]) 100\]

where
SBHReadBuffered:
\[\text{buffered-val} \ sb \ a = \text{Some} \ v\]
\[\implies \text{Read} \ \text{volatile} \ a \ t \ # \text{ is} \ \emptyset, \emptyset, m, D, O, R, S) \rightarrow_{s bh} \text{is} \ \emptyset, \emptyset, m, D, O, R, S, Sb[\text{Read}_{sb} \ \text{volatile} \ a \ t \ v], m, D, O, R, S)\]

SBHReadUnbuffered:
\[\text{buffered-val} \ sb \ a = \text{None}\]
\[\implies \text{Read} \ \text{volatile} \ a \ t \ # \text{ is} \ \emptyset, \emptyset, m, D, O, R, S) \rightarrow_{s bh} \text{is} \ \emptyset, \emptyset, m, D, O, R, S, Sb[\text{Read}_{sb} \ \text{volatile} \ a \ t \ (m \ a)], m, D, O, R, S)\]

SBHWriteNonVolatile:
\[\text{Write} \ \text{False} \ a \ (D, f) \ A \ L \ R \ W \# \text{is} \ \emptyset, \emptyset, sb, m, D, O, R, S) \rightarrow_{s bh} \text{is} \ \emptyset, \emptyset, sb[\text{Write}_{sb} \ \text{False} \ a \ (D, f) (f \ \emptyset) \ A \ L \ R \ W], m, D, O, R, S)\]

SBHWriteVolatile:
\[\text{Write} \ \text{True} \ a \ (D, f) \ A \ L \ R \ W \# \text{is} \ \emptyset, \emptyset, sb, m, D, O, R, S) \rightarrow_{s bh} \text{is} \ \emptyset, \emptyset, sb[\text{Write}_{sb} \ \text{True} \ a \ (D, f) (f \ \emptyset) \ A \ L \ R \ W], m, D, O, R, S)\]

SBHFence:
\[\text{Fence} \ # \text{ is} \ \emptyset, [], m, D, O, R, S) \rightarrow_{s bh} \text{is} \ \emptyset, [], m, False, O, Map.empty, S)\]

SBHRMWReadOnly:
\[\neg \text{cond} \ (\theta (t \mapsto m \ a)) \implies \text{RMW} \ a \ t \ (D, f) \text{ cond ret} A \ L \ R \ W \# \text{ is} \ \emptyset, \emptyset, m, D, O, R, S) \rightarrow_{sb} \text{is} \ \emptyset, \emptyset, m, D, O, R, S, Sb\]
SBHRMWW:

\[(\cond ((\delta(t\rightarrow m a)))] \Rightarrow (\RMW a t (D,f) \cond ret A L R W\# is, \emptyset, [], m, D, O, R, S) \rightarrow_{sbb}

(is, \delta(t\rightarrow ret (m a)) ((f(\delta(t\rightarrow m a))))), [], m(a:= f(\delta(t\rightarrow m a))), False, O \cup A - R, Map.empty, S \oplus W R \ominus A L)

SBHGHOST:

\[(\Ghost A L R W\# is, \emptyset, sb, m, D, O, R, S) \rightarrow_{sbb}

(is, \emptyset, sb\@[\Ghost sb A L R W], m, D, O, R, S)


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 S O [] = True
| non-volatile-owned-or-read-only pending-write S O (x#xs) =
  (case x of
   Read_{sb} volatile a t v \Rightarrow
   (\neg volatile \rightarrow pending-write \rightarrow (a \in O \lor a \in read-only S)) \land
   non-volatile-owned-or-read-only pending-write S O 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 S O 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 S O xs)

primrec acquired :: bool \Rightarrow 'a memref list \Rightarrow addr set \Rightarrow addr set

where
acquired pending-write [] A = (if pending-write then A else { })
| acquired pending-write (x#xs) A =
  (case x of
   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 ⊕\sb W R ⊆\sb A L)) else share xs S)
| Ghost\sb A L R W ⇒ share xs (S ⊕\sb W R ⊆\sb 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 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: 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 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 ∈ acquired pending-write xs A
assumes sub: A ⊆ B
shows x ∈ acquired pending-write xs B
using acquired-mono [OF sub, of pending-write xs] x-in
by blast

lemma acquired-no-pending-write:\A A B. acquired False xs A = acquired False xs B
by (induct xs) (auto split: memref.splits)

lemma acquired-no-pending-write-in:
x ∈ acquired False xs A ⇒ x ∈ acquired False xs B
apply (subst acquired-no-pending-write)
apply auto
done
done

lemma acquired-pending-write-mono-in: \A A B. x ∈ acquired False xs A ⇒ x ∈ acquired True xs B
apply (induct xs)
apply (auto split: memref.splits if-split-asm intro: acquired-mono-in)
done

lemma acquired-append: \A pending-write. acquired pending-write (xs@ys) A =
acquired (pending-write \or outstanding-refs is-volatile-Write_{s_b} xs \neq \{\}) ys (acquired pending-write xs A)
apply (induct xs)
apply (auto split: memref.splits intro: acquired-no-pending-write-in)
done

lemma acquired-take-drop:
acquired (pending-write \or outstanding-refs is-volatile-Write_{s_b} (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 (((takeWhile P xs)@(dropWhile P xs)) A)
  by simp
  also
from acquired-append [where xs=(takeWhile P xs) and ys=(dropWhile P xs)]
have ... = acquired (pending-write \or outstanding-refs is-volatile-Write_{s_b} (takeWhile P xs) \neq \{\})
  (dropWhile P xs) (acquired pending-write (takeWhile P xs) A)
  by simp
finally show ?thesis
  by simp
qed

132
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
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
applyclarsimp
applyclarsimp
applyclarsimp
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
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: \( \forall A \ B. \ acquired-reads \ False \ xs \ A = acquired-reads \ False \ xs \ B \)
by (induct xs) (auto split: memref.splits)

lemma acquired-reads-no-pending-write-in:
  x ∈ acquired-reads False xs A \implies x ∈ acquired-reads False xs B
  apply (subst acquired-reads-no-pending-write)
  apply blast
done

lemma acquired-reads-pending-write-mono:
  \( \forall A. \ acquired-reads \ False \ xs \ A \subseteq acquired-reads \ True \ xs \ A \)
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: \( \forall pending-write A. \ acquired-reads \ pending-write \ (xs@ys) \ A = acquired-reads \ pending-write \ xs \ A \cup acquired-reads \ (pending-write \lor \ (outstanding-refs \ is-volatile-Write_{ab} \ xs \not= \{\})) \ ys \) (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_{ab} volatile a sop v A L R W)
  show ?thesis
  proof (cases volatile)
    case False
    show ?thesis
  using Cons.hyps
by (auto simp add: Write_{ab} False)
next

134
case True
  
show ?thesis

using Cons.hyps

by (auto simp add: Write\textsubscript{sb} True)

qed

done

case (Read\textsubscript{sb} volatile a t v)

show ?thesis

proof (cases volatile)
  
  case False
  
  show ?thesis

using Cons.hyps

by (auto simp add: Read\textsubscript{sb} False)

qed

done

case True

show ?thesis

using Cons.hyps

by (auto simp add: Read\textsubscript{sb} True)

qed

done

case Prog\textsubscript{sb}

with Cons.hyps show ?thesis

by auto

done

case (Ghost\textsubscript{sb} 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: Ghost\textsubscript{sb})

qed

done

lemma in-acquired-reads-no-pending-write-outstanding-write:

∀ A. a ∈ acquired-reads False xs A ⇒ outstanding-refs (is-volatile-Write\textsubscript{sb}) xs ≠ {}

apply (induct xs)

apply simp

apply (auto split: memref.splits)

apply auto

done

lemma augment-read-only-mono: read-only S ⊆ read-only S' ⇒
  
read-only (S ⊕ W R) ⊆ read-only (S' ⊕ W R)

by (auto simp add: augment-shared-def read-only-def)

lemma restrict-read-only-mono: read-only S ⊆ read-only S' ⇒
  
read-only (S ⊓ A L) ⊆ read-only (S' ⊓ A L)

  apply (clarsimp simp add: restrict-shared-def read-only-def split: option.splits)

apply (clarsimp simp add: restrict-shared-def split: option.splits)

apply blast

apply fastforce

done
lemma share-read-only-mono: \( \forall S S'. \text{read-only } S \subseteq \text{read-only } S' \Rightarrow \text{read-only } (\text{share sb } S) \subseteq \text{read-only } (\text{share sb } S') \)

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 (read-only \( S \subseteq \text{read-only } S' \))
      from augment-read-only-mono [OF this]
      have read-only (\( S \oplus W \)) \( \subseteq \text{read-only } (S' \oplus W) \).
      from restrict-read-only-mono [OF this, of A L]
      have read-only (\( S \oplus W \oplus A \)) \( \subseteq \text{read-only } (S' \oplus W \oplus A) \).
      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 (read-only \( S \subseteq \text{read-only } S' \))
  from augment-read-only-mono [OF this]
  have read-only (\( S \oplus W \)) \( \subseteq \text{read-only } (S' \oplus W) \).
  from restrict-read-only-mono [OF this, of A L]
  have read-only (\( S \oplus W \oplus A \)) \( \subseteq \text{read-only } (S' \oplus W \oplus A) \).
  from Cons.hyps [OF this]
  show ?thesis
  by (clarsimp simp add: Ghost sb)
  qed
qede

defined

lemma non-volatile-owned-or-read-only-append:
\( \land O S \) pending-write. non-volatile-owned-or-read-only pending-write \( S O \) (xs@ys)
\( = (\text{non-volatile-owned-or-read-only pending-write } S O \hspace{1em} xs \wedge \hspace{1em}) \)
non-volatile-owned-or-read-only (pending-write \lor outstanding-refs
is-volatile-Write sb xs \neq \{\})

apply (induct xs)
apply (auto split: memref.splits)
done

lemma non-volatile-owned-or-read-only-mono:
\\bigwedge 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:
\\bigwedge 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 \oplus 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

137
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:
\( \forall O S. \text{non-volatile-owned-or-read-only True } S \cup O \text{ xs} \implies \text{non-volatile-owned-or-read-only False } S \cup O \text{ xs} \)
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\(_{sb}\):
\( \forall A. (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{-is-volatile-Write}_{sb}) \text{ sb}) \text{ A}) \subseteq A \cup \text{ all-acquired } (\text{takeWhile } (\text{Not } \circ \text{-is-volatile-Write}_{sb}) \text{ sb}) \)
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 \circ \text{-is-volatile-Write}_{sb}) \text{ sb}) \text{ A} = {}
apply (induct sb)
apply simp
lemma outstanding-refs-takeWhile-opposite: outstanding-refs P (takeWhile (Not o P) xs) = {}
apply (induct xs)
apply auto
done

done

lemma no-outstanding-volatile-Write\sb-acquired:
outstanding-refs is-volatile-Write\sb sb = {} \implies acquired False sb A = {}
apply (induct sb)
apply simp
subgoal for a sb
  by (case-tac a) auto
done

done

lemma acquired-all-acquired:\A pending-write A. acquired pending-write xs A \subseteq A \cup all-acquired xs
apply (induct xs)
apply (auto split: memref.splits)
done

lemma acquired-all-acquired-in: x \in acquired pending-write xs A \implies x \in A \cup all-acquired xs
using acquired-all-acquired
by blast

primrec sharing-consistent:: shared \Rightarrow owns \Rightarrow 'a memref list \Rightarrow bool
where
  sharing-consistent S O [] = True
  | sharing-consistent S O (r#rs) =
    (case r of
      Write\sb volatile - - - A L R W \Rightarrow
        (if volatile then A \subseteq dom S \cup O \land L \subseteq A \land A \cap R = {} \land R \subseteq O \land
           sharing-consistent (S \oplus W R \ominus A L) (O \cup A - R) rs
         else sharing-consistent S O rs)
      | Ghost\sb A L R W \Rightarrow A \subseteq dom S \cup O \land L \subseteq A \land A \cap R = {} \land R \subseteq O \land
           sharing-consistent (S \oplus W R \ominus A L) (O \cup A - R) rs
      | _ \Rightarrow sharing-consistent S O rs)

lemma sharing-consistent-all-acquired:
\A S O. sharing-consistent S O sb \implies all-acquired sb \subseteq dom S \cup O
proof (induct sb)
  case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
  case (Write a volatile - - - A L R W)
  show ?thesis
  proof (cases volatile)
  case False
    with Cons Write show ?thesis by auto
  next
  case True
  from Cons.hyps [where $S = (S \oplus W R \ominus A) \land O = (O \cup A - R)$] Cons.prems
  show ?thesis
  by (auto simp add: Write 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 $S = (S \oplus W R \ominus A) \land O = (O \cup A - R)$] Cons.prems show ?thesis by auto
  qed
  qed

lemma sharing-consistent-append:
\[\forall S, O. \text{ sharing-consistent } S \cdot O \cdot (xs \cdot ys) =\]
\[
\text{(sharing-consistent } S \cdot O \cdot xs \land \text{ sharing-consistent } (\text{share } xs \cdot S) \land \text{(acquired True } xs \cdot O) \land ys)\]
apply (induct xs)
apply (auto split: memref.splits)
done

primrec read-only-reads :: owns => 'a memref list => addr set
where
read-only-reads O [] = {}
| read-only-reads O (x#xs) =
  (case x of
    Read a volatile - t v ⇒ (if \neg volatile \land a \notin O
    then insert a (read-only-reads O xs)
    else read-only-reads O xs)
  | Write a volatile - - - A L R W ⇒
    (if volatile then read-only-reads (O \cup A - R) xs
    else read-only-reads O xs )
  | Ghost a A L R W ⇒ read-only-reads (O \cup A - R) xs
  | - ⇒ read-only-reads O xs)

lemma read-only-reads-append:
\[\forall O. \text{ read-only-reads } O \cdot (xs \cdot ys) =\]
\[
\text{read-only-reads } O \cdot xs \cup \text{read-only-reads } (\text{acquired True } xs \cdot O) \cdot ys\]
apply (induct xs)
apply simp
subgoal for a xs O
  by (case-tac a) auto
done

lemma read-only-reads-antimono:
\[ \forall O, O'. \quad O \subseteq O' \implies \text{read-only-reads } O' \text{ sb } \subseteq \text{read-only-reads } O \text{ sb} \]
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 a sop v A L R W \Rightarrow (if volatile then non-volatile-writes-unshared (S \oplus_W R \ominus_A L) xs
      else a \notin \text{dom } S \land \text{non-volatile-writes-unshared } S \text{ xs})
    | Ghost sb A L R W \Rightarrow \text{non-volatile-writes-unshared } (S \oplus_W R \ominus_A L) \text{ xs}
    | - \Rightarrow \text{non-volatile-writes-unshared } S \text{ xs})

lemma non-volatile-writes-unshared-append:
\[ \forall S. \quad \text{non-volatile-writes-unshared } S \text{ (xs@ys)} = \]
  = (non-volatile-writes-unshared S xs \land \text{non-volatile-writes-unshared } (\text{share } xs S) \text{ ys})
apply (induct xs)
apply (auto split: memref.splits)
done

lemma non-volatile-writes-unshared-antimono:
\[ \forall S, S'. \quad \text{dom } S \subseteq \text{dom } S' \implies \text{non-volatile-writes-unshared } S' \text{ xs} \]
\[ \Rightarrow \text{non-volatile-writes-unshared } S \text{ xs} \]

```plaintext
apply \( \text{induct } \text{xs} \)
apply \( \text{simp} \)

subgoal for a \( \text{xs} \) \( S \) \( S' \)
apply \( \text{case-tac a} \)
apply \( \text{(clarsimp split: if-split-asm)} \)
subgoal for volatile a1 D f v A L R W
apply \( \text{drule-tac } C=R \text{ in } \text{augment-mono-aux} \)
apply \( \text{drule-tac } C=L \text{ in } \text{restrict-mono-aux} \)
apply \( \text{blast} \)
done
```
subgoal for a xs S S'
apply (case-tac a)
apply (clarsimp split: if-split-asm)
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)
apply (fastforce dest: read-only-mono shared-leD)
done
apply (fastforce dest: read-only-mono shared-leD)
apply fastforce
apply fastforce
apply (clarsimp)
subgoal for A L R W
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)
apply (fastforce dest: read-only-mono shared-leD)
done
done
done

locale outstanding-non-volatile-ref-owned-or-read-only =
fixes S::shared
fixes ts::('p, 'p store-buffer, bool, owns, rels) thread-config list
assumes outstanding-non-volatile-ref-owned-or-read-only:
\[ \forall i \text{ is } O \ R \ D \ \theta \ sb \ p. \]
\[ [ i < \text{length ts}; \text{ts!i} = (p, i, \theta, sb, D, O, R) ] \]
\[ \implies \]
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:
\[ \forall i \ p_i \ i \text{ is } O_i \ R_i \ D_i \ \theta_i \ sb_i \ j \ p_j \ i \text{ is } O_j \ R_j \ D_j \ \theta_j \ sb_j. \]
\[ [ i < \text{length ts}; j < \text{length ts}; i \neq j; \]
\[ \text{ts!i} = (p_i, i, \theta_i, sb_i, D_i, O_i, R_i); \text{ts!j} = (p_j, i, \theta_j, sb_j, D_j, O_j, R_j) ] \]
\[ \implies \]
(\( O_j \cup \text{all-acquired sb}_j \)) \cap outstanding-ref is-volatile-Write sb sb_i = {}
locale ownership-distinct =
fixes ts::('p', 'p store-buffer, bool, owns,rels) thread-config list
assumes ownership-distinct: 
\[ \forall i \ j \ p \ i is \ O_i \ R_i \ D_i \ \forall \ j \ sb_i \ i is \ j \ O_j \ R_j \ D_j \ \forall \ j \ sb_j. \]
\[ [i < \text{length ts}; j < \text{length ts}; i \neq j;\]
\[ \text{tsli} = (p_i, is_i, \theta_i, sb_i, D_i, O_i, R_i); \text{tslj} = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j); \]
\[ \implies (O_i \cup \text{all-acquired sb}_i) \cap (O_j \cup \text{all-acquired sb}_j) = \{} \]

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:
\[ \forall i \ p \ is \ O \ R \ D \ \forall \ sb. \]
\[ [i < \text{length ts}; tsli = (p, is, \theta, sb, D, O, R) \]
\[ \implies \non-volatile-writes-unshared \ S \ sb \]

locale sharing-consis =
fixes S::shared and ts::('p', 'p store-buffer, bool, owns, rels) thread-config list
assumes sharing-consis:
\[ \forall i \ p \ is \ O \ R \ D \ \forall \ sb. \]
\[ [i < \text{length ts}; tsli = (p, is, \theta, sb, D, O, R) \]
\[ \implies \sharing-consistent \ S \ O \ sb \]

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:
\[ \forall i \ p \ is \ O \ R \ D \ \forall \ sb. \]
\[ [i < \text{length ts}; tsli = (p, is, \theta, sb, D, O, R) \]
\[ \implies \no-write-to-read-only-memory \ S \ sb \]

locale valid-sharing =
outstanding-non-volatile-writes-unshared +
sharing-consis +
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:
\[ i \land p \in O \ R \ D \ \not\in sb. \]
\[ i < \text{length } ts; ts[i] = (p, is, \theta, sb, D, O, R); \]
read-only-reads (acquired True (takeWhile (Not is-volatile-Write sb) sb') O')
\[ (\text{dropWhile (Not is-volatile-Write sb)} sb) \subseteq \text{read-only-reads (acquired True (takeWhile (Not is-volatile-Write sb) sb)} O) \]
\[ (\text{dropWhile (Not is-volatile-Write sb)} sb); \]
\[ O' \cup \text{all-acquired sb'} \subseteq O \cup \text{all-acquired sb} \] \[ \Rightarrow \]
\[ \text{read-only-reads-unowned (ts[i := (p', is', \theta', sb', D', O', R')])} \]
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) \[ \Rightarrow \]
outstanding-non-volatile-refs-owned-or-read-only S ts
by (force 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) \[ \Rightarrow \]
outstanding-volatile-writes-unowned-by-others ts
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) \[ \Rightarrow \]
read-only-reads-unowned (ts)
apply (clarsimp simp add: read-only-reads-unowned-def)
apply fastforce
done

lemma ownership-distinct-tl:
assumes dist: ownership-distinct (t#ts)

145
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)

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) ⇒
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) $\implies$
no-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:
$\forall O S$. pending-write. [non-volatile-owned-or-read-only pending-write $S$ $O$ sb]
$\implies$
outstanding-refs is-non-volatile-$\text{Write}_{sb}$ $sb \subseteq O \cup$ all-acquired sb
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write$_{sb}$ volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True
from Cons.hyps [of True ($S \uplus W$ 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.hyps [of pending-write ($S \uplus W$ R $\ominus_A$ L) ($O \cup A - R$)] Cons.prems
show ?thesis
by (auto simp add: Ghost$_{sb}$)

147
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, o, sb, D, O, R)
shows outstanding-refs is-non-volatile-Write sb ⊆ O ∪ 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:
∀O S. [non-volatile-owned-or-read-only True S O sb; sharing-consistent S O sb]
⇒
outstanding-refs is-non-volatile-Read sb ⊆ O ∪ all-acquired sb ∪ 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 ⊕ 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: Write sb True )

from Cons.hyps [OF non-vol consis']
have hyp: outstanding-refs is-non-volatile-Read 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: Write sb True )
apply (drule (1) rev-subsetD)
apply (auto simp add: in-read-only-consvs 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-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-volatile-ovs]
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

lemma non-volatile-reads-acquired-or-read-only-reads:
\( \forall O S \) 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

from Cons.prems obtain non-volatile-owned-or-read-only True \( S ⊕ W R ⊕ A L \) \( O ⊔ A − R \) sb
by (clarsimp simp add: Write sb True )
from Cons.hyps [OF non-vol ]
have hyp: outstanding-refs is-non-volatile-Read_{sb} sb

    ⊆ \mathcal{O} \cup A - R \cup \text{all-acquired sb} \cup \text{read-only-reads} (\mathcal{O} \cup A - R) \sb.

then

  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
  qed
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) (\mathcal{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

    ⊆ \mathcal{O} \cup A - R \cup \text{all-acquired sb} \cup \text{read-only-reads} (\mathcal{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:
\\[ \land_{\mathcal{O}} S \text{ pending-write}. \ [\text{non-volatile-owned-or-read-only pending-write} S \mathcal{O} \sb]\n\Longrightarrow

\text{outstanding-refs} (\text{Not} \circ \text{is-volatile}) \sb \subseteq \mathcal{O} \cup \text{all-acquired sb} \cup \text{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
      from Cons.hyps [of True (S \oplus_W R \ominus_A L) (\mathcal{O} \cup A - R)] Cons.prems
      show ?thesis
  qed
  qed


150
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
qed

lemma no-unacquired-write-to-read-only:
\( \forall S O. [\text{no-write-to-read-only-memory } S \text{ sb;sharing-consistent } S O \text{ sb;}
\forall a \in \text{read-only } S; \forall a \notin (O \cup \text{all-acquired sb})] \implies a \notin \text{outstanding-refs is-Write sb} \)

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 } L-A: L \subseteq A \text{ and } A-R: A \cap R = \{\} \text{ and } R-owns:
  R \subseteq O \text{ and }
  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 }
  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 )

  from a'-notin a-ro have neq-a-a': a\#a'
by blast

  from a-A a-all-acq a-owns
have a-notin': a ∉ O ∪ A − R ∪ all-acquired sb
by auto
from a-ro L-A a-A R-owns a-owns
have a ∈ read-only (S ⊕W R ⊕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 ∉ outstanding-refs is-Write_{ab} sb.
with neq-a-a'
show ?thesis
by (clarsimp simp add: Write_{ab} True)

next
  case False with Cons
  show ?thesis
  by (auto simp add: Write_{ab} 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 ⊕W R ⊕A L) 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 S and
  a-A: a ∉ A and a-all-acq: a ∉ all-acquired sb and a-owns: a ∉ O
  by ( simp add: Ghost_{ab} )

from a-A a-all-acq a-owns
have a-notin': a ∉ O ∪ A − R ∪ all-acquired sb
by auto
from a-ro L-A a-A R-owns a-owns
have a ∈ read-only (S ⊕W R ⊕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 ∉ outstanding-refs is-Write_{ab} sb.
then
show ?thesis
by (clarsimp simp add: Ghost_{ab})
qed

qed
lemma read-only-reads-read-only:
\[ \forall S. \text{[non-volatile-owned-or-read-only True } S \cap O \cap \text{ sb]} \]
\[ \implies \text{read-only-reads } O \cap \text{ sb } \subseteq \cap O \cup \text{ all-acquired sb } \cup \text{ read-only } S \]

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: \text{non-volatile-owned-or-read-only True } (S \oplus (W R \ominus A) L) \cap \cap O \cup A \cap \text{ sb and} \\ A\text{-shared-onws: } A \subseteq \text{ dom } S \cup O \text{ and } L\text{-A: } L \subseteq A \text{ and } A\text{-R: } A \cap R = \{\} \text{ and } R\text{-owns: } R \subseteq O \cap \text{ and} \\ \text{consis': sharing-consistent } (S \oplus (W R \ominus A L) \cap (O \cup A \cap \text{ R}) \cap \text{ sb by (clarsimp simp add: Write sb True )}

      from Cons.hyps [OF non-vol consis']
      have hyp: \text{read-only-reads } (O \cup A \cap \text{ R}) \cap \text{ sb }

      \subseteq O \cup A \cap \text{ R } \cup \text{ all-acquired sb } \cup \text{ read-only } (S \oplus R \ominus A L).

      
      \{ 

      fix a' 
      assume a'-in: a' \in \text{read-only-reads } (O \cup A \cap \text{ R}) \cap \text{ sb }
      assume a'-unowned: a' \notin O
      assume a'-unacq: a' \notin \text{ all-acquired sb }
      assume a'-A: a' \notin A
      have a' \in \text{read-only } S
      proof 
        from a'-in hyp a'-unowned a'-unacq a'-A
        have a' \in \text{read-only } (S \oplus (W R \ominus A L)

        by auto

        with L-A R\text{-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: Writeab)
qed
next
case \text{Read}_{sb} with Cons show \ ?thesis
by auto
next
case \text{Prog}_{sb} with Cons show \ ?thesis
by auto
next
case \text{Ghost}_{sb} A L R W
from \text{Cons.prems} obtain non-vol: non-volatile-owned-or-read-only \text{True} (S \oplus W R \ominus A L) (O \cup A - R) \text{ 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}
consis': sharing-consistent (S \oplus W R \ominus A L) (O \cup A - R) \text{ sb}
by (clarsimp simp add: Ghost_{sb})
from Cons.hyps \[OF \text{ non-vol consis'}\]
have hyp: \text{read-only-reads} (O \cup A - R) \text{ sb}
\subseteq O \cup A - R \cup \text{all-acquired sb} \cup \text{read-only} (S \oplus W R \ominus A L).

\{
fix a'
assume a'-in: a' \in \text{read-only-reads} (O \cup A - R) \text{ sb}
assume a'-unowned: a' \notin O
assume a'-unacq: a' \notin \text{all-acquired sb}
assume a'-A: a' \notin A
have a' \in \text{read-only } S
proof –
from a'-in hyp a'-unowned a'-unacq a'-A
have a' \in \text{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: Ghost_{sb} simp del: o-apply)
apply force
done

154
lemma no-unacquired-write-to-read-only-reads:
\[ S O . \ \text{[no-write-to-read-only-memory \( S \) \( sb \)}; \non-volatile-owned-or-read-only \text{True} \( S O \) \( sb \); \text{sharing-consistent} \( S O \) \( sb \);
a \in \text{read-only-reads} \( O sb \); a \notin (O \cup \text{all-acquired} \( sb \)]
\[ \implies a \notin \text{outstanding-refs is-Write}\_sb \]

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: \text{no-write-to-read-only-memory (} S \oplus W R \ominus A L) sb
and
non-vol: \text{non-volatile-owned-or-read-only True} \( (S \oplus W R \ominus A L) (O \cup A - R) sb \) and
A-shared-own: 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’: \text{sharing-consistent (} S \oplus W R \ominus A L) (O \cup A - R) sb \) and
a-own: a \in \text{read-only-reads (} O \cup A - R) sb \) and
a-A: a \notin A and a-all-acq: a \notin \text{all-acquired} \( sb \) and a-owns: a \notin O and
a’-notin: a’ \notin \text{read-only} \( S \)
by ( simp add: Write\_sb True )
from read-only-reads-read-only \text{[OF non-vol consis’]} a-own 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\neq 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 no-wrt non-vol consis’ a-own a-notin’
have a \notin \text{outstanding-refs is-Write}\_sb sb.
then
show ?thesis
using neq-a-a’
by (auto simp add: Write\_sb True)
next
case False with Cons
show ?thesis
qed
qed
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 (auto simp add: Read_sb)
next
case False
note non-volatile = this
from Cons.prems obtain no-wrt': no-write-to-read-only-memory S sb and
consis'sharing-consistent S O sb and
a-in: a ∈ (if a' ∉ O then insert a' (read-only-reads O sb)
else read-only-reads O sb) and
a'-owns-shared: a' ∈ O ∨ a' ∈ read-only S and
non-vol': non-volatile-owned-or-read-only True S O sb and
a-owns: a ∉ O ∪ all-acquired sb
by (clarsimp simp add: Read_sb False)

show ?thesis
proof (cases a' ∈ O)
case True
with a-in have a ∈ read-only-reads O 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 O 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 O sb) by auto
from Cons.hyps [OF no-wrt' non-vol' consis' this a-owns]
show ?thesis
by (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 sb
by auto
then show ?thesis
by (simp add: Read_sb True)
qed
qed
qed

next
case Prog\sb with Cons
show \textit{?thesis}
  by (auto)
next
case (Ghost\sb 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
  non-vol: non-volatile-owned-or-read-only True \((S \oplus W R \ominus A L)\) \((O \cup A - R)\) \sb
  A-shared-ons: \(A \subseteq \text{dom } S \cup O\) \textit{and} \(L-A: L \subseteq A\) \textit{and} \(A-R: A \cap R = \{\}\) \textit{and} \(R\)-owns:
  \(R \subseteq O\) \textit{and}
  consis': sharing-consistent \((S \oplus W R \ominus A L)\) \((O \cup A - R)\) \sb
  a-ro: \(a \in \text{read-only-reads } (O \cup A - R)\) \sb
  a-A: \(a \notin A\) \textit{and} \(a\)-all-acq: \(a \notin \text{all-acquired}\) \sb
  a-owns: \(a \notin O\)
  by ( simp add: Ghost\sb )

from read-only-reads-read-only [OF non-vol 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

from a-A a-all-acq a-owns
have a-notin': \(a \notin O \cup A - R\) \cup all-acquired sb
  by auto

from Cons.hyps [OF no-wrt non-vol consis’ a-ro a-notin’] have \(a \notin \text{outstanding-refs is-Write}_sb sb\).

then
show \textit{?thesis}
  by (auto simp add: Ghost\sb)
qed
qed

lemma no-unacquired-write-to-read-only’’:
  \textbf{assumes} no-wrt: no-write-to-read-only-memory \(S\) \sb
  \textbf{assumes} consis: sharing-consistent \(S\ O\ sb\)
  \textbf{shows} read-only \(S \cap \text{outstanding-refs is-Write}_sb sb \subseteq O \cup \text{all-acquired}\ sb\)
  \textbf{using} no-unacquired-write-to-read-only [OF no-wrt consis]
  \textbf{by} auto

lemma no-unacquired-volatile-write-to-read-only:
  \textbf{assumes} no-wrt: no-write-to-read-only-memory \(S\) \sb
  \textbf{assumes} consis: sharing-consistent \(S\ O\ sb\)
  \textbf{shows} read-only \(S \cap \text{outstanding-refs is-volatile-Write}_sb sb \subseteq O \cup \text{all-acquired}\ sb\)

\textbf{proof}
  \textbf{have} outstanding-refs is-volatile-Write\sb sb \subseteq outstanding-refs is-Write\sb sb
  \textbf{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 $S$ $O$ sb
  shows read-only $S$ $\cap$ outstanding-refs is-non-volatile-Write sb sb $\subseteq$ $O$ $\cup$ 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-reads' :
  assumes no-wrt: no-write-to-read-only-memory $S$ sb
  assumes non-vol: non-volatile-owned-or-read-only True $S$ $O$ sb
  assumes consis: sharing-consistent $S$ $O$ sb
  shows read-only-reads $O$ sb $\cap$ outstanding-refs is-Write sb sb $\subseteq$ $O$ $\cup$ 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 $S$ $O$ sb
  assumes consis: sharing-consistent $S$ $O$ sb
  shows read-only-reads $O$ sb $\cap$ outstanding-refs is-volatile-Write sb sb $\subseteq$ $O$ $\cup$ 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 $S$ $O$ sb
  assumes consis: sharing-consistent $S$ $O$ sb
  shows read-only-reads $O$ sb $\cap$ outstanding-refs is-non-volatile-Write sb sb $\subseteq$ $O$ $\cup$ all-acquired sb
proof –
  from outstanding-refs-subsets
have outstanding-refs is-non-volatile-Write sb ⊆ outstanding-refs is-Write 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 ∈ set (dropWhile P xs) ⟹ x ∈ set xs
by (induct xs) (auto split: if-split-asm)

lemma outstanding-refs-takeWhileD:
x ∈ outstanding-refs P (takeWhile P′ sb) ⟹ x ∈ outstanding-refs P sb
using outstanding-refs-takeWhile
by blast

lemma outstanding-refs-dropWhileD:
x ∈ outstanding-refs P (dropWhile P′ sb) ⟹ x ∈ outstanding-refs P sb
by (auto dest: set-dropWhileD simp add: outstanding-refs-conv)

lemma dropWhile-ConsD: dropWhile P xs = y#ys ⟹ ¬ 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
⟹ non-volatile-owned-or-read-only True

(share (takeWhile (Not ◦ is-volatile-Write sb) sb) S)
(acquired True (takeWhile (Not ◦ is-volatile-Write sb) sb) O)
(dropWhile (Not ◦ is-volatile-Write sb) sb)
using non-volatile-owned-or-read-only-append [of False S O (takeWhile (Not ◦ is-volatile-Write sb) sb)
(dropWhile (Not ◦ is-volatile-Write sb) sb)]
apply (cases outstanding-refs is-volatile-Write sb = { })
apply (clarsimp simp add: outstanding-refs-conv)
apply (clarsimp simp add: outstanding-refs-conv)
apply (clarsimp simp add: outstanding-refs-conv)
done

lemma read-only-share: \( \bigwedge S O. \)
sharing-consistent S O sb ⟹ 
read-only (share sb S) ⊆ read-only S ∪ O ∪ all-acquired sb

159
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.prems obtain
A-shared-owns: A ⊆ dom $S \cup O$ and L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ 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 consis']
have read-only (share sb $(S \oplus W R \ominus A L)$)
  ⊆ read-only $(S \oplus W R \ominus A L) \cup (O \cup A - R) \cup$ all-acquired sb
by auto
also from A-shared-owns L-A R-owns A-R
have read-only $(S \oplus W R \ominus A L) \cup (O \cup A - R) \cup$ all-acquired sb ⊆
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
show ?thesis
  by (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 obtain
A-shared-owns: A ⊆ dom $S \cup O$ and L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns: R ⊆ O and
consis': sharing-consistent $(S \oplus W R \ominus A L) (O \cup A - R)$ sb
by (clarsimp simp add: Ghost sb)
from Cons.hyps [OF consis']
have read-only (share sb $(S \oplus W R \ominus A L)$)
  ⊆ read-only $(S \oplus W R \ominus A L) \cup (O \cup A - R) \cup$ all-acquired sb
by auto
also from A-shared-owns L-A R-owns A-R
have read-only $(S \oplus W R \ominus A L) \cup (O \cup A - R) \cup$ all-acquired sb ⊆
read-only $S \cup \mathcal{O} \cup (A \cup \text{all-acquired sb})$

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

finally

show ?thesis

by (simp add: Ghost_sb)

qed

qed

lemma (in 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: $\text{ts}!i = (p_i, \text{is}_i, \theta_i, \text{sb}_i, D_i, O_i, R_i)$
assumes jth: $\text{ts}!j = (p_j, \text{is}_j, \theta_j, \text{sb}_j, D_j, O_j, R_j)$

shows $\text{outstanding-refs is-Write}_{\text{sb}}\ (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) \ \text{sb}_i) \cap \\
\text{outstanding-refs is-non-volatile-Read}_{\text{sb}}\ (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) \ \text{sb}_j) = \{\}$

proof –

let ?take-j = (takeWhile (Not $\circ$ is-volatile-Write$_{\text{sb}}$) sb$_j$)
let ?drop-j = (dropWhile (Not $\circ$ is-volatile-Write$_{\text{sb}}$) sb$_j$)

let ?take-i = (takeWhile (Not $\circ$ is-volatile-Write$_{\text{sb}}$) sb$_i$)
let ?drop-i = (dropWhile (Not $\circ$ is-volatile-Write$_{\text{sb}}$) sb$_i$)

note nvo-i = outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ith]
note nvo-j = outstanding-non-volatile-refs-owned-or-read-only [OF j-bound jth]
note nro-i = no-outstanding-write-to-read-only-memory [OF i-bound ith]
with no-write-to-read-only-memory-append [of $S$ ?take-i ?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 [OF j-bound jth]
with no-write-to-read-only-memory-append [of $S$ ?take-j ?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 [OF i-bound j-bound neq-i-j ith jth]
have dist: $(\mathcal{O}_j \cup \text{all-acquired sb}_j) \cap \text{outstanding-refs is-volatile-Write}_{\text{sb}} \ \text{sb}_i = \{\}.$
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 $S \ \mathcal{O}_j \ \text{sb}_j$.
with sharing-consistent-append [of $S \ \mathcal{O}_j$ ?take-j ?drop-j]
obtain

161
consis-take-j: sharing-consistent $S \ O_j$ ?take-j and 
by simp 

from sharing-consis [OF i-bound ith] 
have consis-i: sharing-consistent $S \ O_i$ sbi. 
with sharing-consistent-append [of $S \ O_i$ ?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$_{sb}$ sbi 
    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$_{sb}$ sbj 
    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. 

    from non-volatile-reads-acquired-or-read-only-reads [OF nvo-drop-j | x-in-drop-j 
    acquired-takeWhile-non-volatile-Write$_{sb}$ [of sbj $O_j$] 
    have x-j: $x \in O_j \cup$ all-acquired sbj $\cup$ 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$_{sb}$ sbi 
  using outstanding-refs-append [of is-volatile-Write$_{sb}$ ?take-i ?drop-i] 
  by auto 

from outstanding-volatileWrites-unowned-by-others [OF i-bound j-bound neq-i-j ith jth] 
have ($O_j \cup$ all-acquired sbj) $\cap$ outstanding-refs is-volatile-Write$_{sb}$ sbi = $\{\}$. 

with x-in-vol-i x-j obtain 
  x-unacq-j: $x \notin O_j \cup$ all-acquired sbj and 
  x-ror-j: $x \in$ read-only-reads (acquired True ?take-j $O_j$) ?drop-j 
  by auto 
from read-only-reads-unowned [OF j-bound i-bound neq-i-j [symmetric] jth ith] x-ror-j 
have $x \notin O_i \cup$ all-acquired sbi 
  by auto 

162
moreover

from read-only-reads-read-only [OF nvo-drop-j consis-drop-j] x-ror-j x-unacq-j
   all-acquired-append [of ?take-j ?drop-j] acquired-takeWhile-non-volatile-Write_{sb \ j}
have x ∈ read-only (share ?take-j S)
   by (auto)

   from read-only-share [OF consis-take-j] this x-unacq-j all-acquired-append [of ?take-j
   ?drop-j]
   have x ∈ read-only S
   by auto

with no-unacquired-write-to-read-only" [OF nro-i consis-i] x-in-i
have x ∈ O_{i \ j} ∪ all-acquired sb_{i \ j}
   by auto

ultimately have False by auto
   }
   moreover
   }
assume x-in-non-vol-drop-i: x ∈ outstanding-refs is-non-volatile-Write_{sb \ i}
  hence x ∈ outstanding-refs is-non-volatile-Write_{sb \ i}
  using outstanding-refs-append [of is-non-volatile-Write_{sb \ i} ?take-i ?drop-i]
  by auto
with non-volatile-owned-or-read-only-outstanding-non-volatile-writes [OF nvo-i]
have x ∈ O_{i \ j} ∪ all-acquired sb_{i \ j} by auto

moreover

with x-j own-dist obtain
  x-unacq-j: x \∉ O_{i \ j} ∪ all-acquired sb_{i \ j} and
  x-ror-j: x ∈ read-only-reads (acquired True ?take-j O_{j \ i}) ?drop-j
by auto
from read-only-reads-unowned [OF j-bound i-bound neq-i-j [symmetric] jth ith] x-ror-j
have x \∉ O_{i \ j} ∪ all-acquired sb_{i \ j}
by auto

ultimately have False
by auto
}
ultimately

show ?thesis
using x-in-drop-i x-in-drop-j
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: \( i < \text{length } ts \)

assumes j-bound: \( j < \text{length } ts \)

assumes neq-i-j: \( i \neq j \)

assumes ith: \( ts!i = (p_i, i_s, i_d, \theta_i, sb_i, D_i, O_i, R_i) \)

assumes jth: \( ts!j = (p_j, i_s, i_d, \theta_j, sb_j, D_j, O_j, R_j) \)

shows outstanding-refs \((\text{is-non-volatile-Write}_{sb_i})\) \((\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb_i}) \text{ sb}_i) \cap \)

(outstanding-refs \((\text{is-volatile-Write}_{sb_i})\) \((\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb_i}) \text{ sb}_j) \cup \)

(outstanding-refs \((\text{is-volatile-Write}_{sb_j})\) \((\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb_j}) \text{ sb}_j) \)

\(= \) \{\} \((\text{is } \text{?non-vol-writes-i } \cap \text{?not-volatile-j } = \{\})\)

proof –

note nro-i = no-outstanding-write-to-read-only-memory \([\text{OF i-bound ith}]\)

note nro-j = no-outstanding-write-to-read-only-memory \([\text{OF j-bound jth}]\)

note nvo-j = outstanding-non-volatile-refs-owned-or-read-only \([\text{OF j-bound jth}]\)

note nvo-i = outstanding-non-volatile-refs-owned-or-read-only \([\text{OF i-bound ith}]\)

from outstanding-volatile-writes-unowned-by-others \([\text{OF i-bound j-bound neq-i-j ith jth}]\)

have \( \text{dist}: (O_j \cup \text{all-acquired sb}_j) \cap \text{outstanding-refs is-volatile-Write}_{sb_i} \text{ sb}_j = \{\}. \)

from outstanding-volatile-writes-unowned-by-others \([\text{OF j-bound i-bound neq-i-j symmetric] jth ith}]\)

have \( \text{dist-j}: (O_i \cup \text{all-acquired sb}_i) \cap \text{outstanding-refs is-volatile-Write}_{sb_j} \text{ sb}_j = \{\}. \)

note own-dist = ownership-distinct \([\text{OF i-bound j-bound neq-i-j ith jth}]\)

from sharing-consis \([\text{OF j-bound jth}]\)

have \( \text{consis-j: sharing-consistent } S \ O_j \text{ sb}_j. \)

from sharing-consis \([\text{OF i-bound ith}]\)

have \( \text{consis-i: sharing-consistent } S \ O_i \text{ sb}_i. \)

let \(?\text{take-j} = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb_j}) \text{ sb}_j)\)

let \(?\text{drop-j} = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb_j}) \text{ sb}_j)\)

{ 
  fix x

  assume x-in-take-i: \( x \in \text{?non-vol-writes-i} \)

  assume x-in-j: \( x \in \text{?not-volatile-j} \)

  from x-in-take-i have x-in-i: \( x \in \text{outstanding-refs } (\text{is-non-volatile-Write}_{sb_i}) \text{ sb}_i \)

  164
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} 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} 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} ?drop-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-convs)
moreover
  all-acquired-append [of ?take-j ?drop-j]
have acquired True ?take-j O_j ∪ all-acquired ?drop-j ⊆ O_j ∪ all-acquired sb_j
  by auto
ultimately
have x ∈ read-only-reads (acquired True ?take-j O_j) ?drop-j
  using x-in-owns-acq-i own-dist
  by auto

  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 ∉ read-only-reads 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 O_{j} ?take-j
by auto

from non-volatile-owned-or-read-only-outstanding-refs [OF this] x-in-j x-notin
have x ∈ O_{j} ∪ all-acquired ?take-j
by (auto simp add: misc-outstanding-refs-convs )
have False
by auto
}
moreover
{
assume x-in-j: x ∈ O_{j} ∪ 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 ≠ j
assumes ith: ts!i = (p_{i}, is_{i}, δ_{i}, sb_{i}, D_{i}, O_{i}, R_{i})
assumes jth: ts!j = (p_{j}, is_{j}, δ_{j}, sb_{j}, D_{j}, O_{j}, R_{j})
shows outstanding-refs (is-non-volatile-Write_{sb}) (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_{i}) ∩
outstanding-refs (Not ◦ is-volatile-Read_{sb}) (dropWhile (Not ◦ is-volatile-Write_{sb}) sb_{j}) = {}
(is ?non-vol-writes-i ∩ ?not-volatile-j = {})
proof –
have outstanding-refs (Not ◦ is-volatile-Read sb) (dropWhile (Not ◦ is-volatile-Write sb) sbj) ⊆ 
  outstanding-refs is-volatile-Write\sb sbj ∪ 
  outstanding-refs is-non-volatile-Write\sb sbj ∪ 
  outstanding-refs is-non-volatile-Read\sb sbj (dropWhile (Not ◦ is-volatile-Write sb) sbj)
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 
  (let (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = ts!i; 
    (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = ts!j 
  in outstanding-refs is-Write\sb sbi ∩ 
  outstanding-refs is-Write\sb (takeWhile (Not ◦ is-volatile-Write sb) sbj) = 
  \{\})
proof – 
{ 
  fix i j p_i is_i O_i R_i D_i \theta_i sb_i p_j is_j O_j R_j D_j \theta_j sb_j 
  assume i-bound: i < length ts 
  assume j-bound: j < length ts 
  assume neq-i-j: i \neq j 
  assume ith: ts!i = (p_i,is_i,\theta_i,\cdot,\cdot,\cdot,\cdot) 
  assume jth: ts!j = (p_j,is_j,\theta_j,\cdot,\cdot,\cdot,\cdot) 
  from outstanding-non-volatile-write-disj [OF j-bound i-bound neq-i-j[ symmetric ] jth ith]
  have outstanding-refs is-Write\sb sbi ∩ 
      outstanding-refs is-Write\sb (takeWhile (Not ◦ is-volatile-Write sb) sbj) = \{}
  apply (clarsimp simp add: outstanding-refs-is-non-volatile-Write\sb-takeWhile-conv) 
  apply (auto simp add: misc-outstanding-refs-convs ) 
  done
} 
thus ?thesis
by (fastforce simp add: Let-def)
qed

fun read-tmps:: 'p store-buffer ⇒ tmp set
where 
  read-tmps [] = {}
| read-tmps (r#rs) = 
  (case r of 
    Read\sb volatile a t v ⇒ insert t (read-tmps rs) 
  | _ ⇒ read-tmps rs)
lemma in-read-tmps-conv:
\((t \in \text{read-tmps } xs) = (\exists \text{volatile } a \text{ v. } \text{Read}_{sb} \text{ volatile } a \ t \ v \in \text{set } xs)\)
by (induct xs) (auto split: memref.splits)

lemma read-tmps-mono: \(\forall \text{ys. set } xs \subseteq \text{set } ys \Rightarrow \text{read-tmps } xs \subseteq \text{read-tmps } ys\)
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_{sb} volatile a t v ⇒ t \not\in (\text{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 \rightarrow
  (\text{case } xs!i \text{ of } \text{Read}_{sb} - - t_i - \Rightarrow \text{case } xs!j \text{ of } \text{Read}_{sb} - - t_j - \Rightarrow 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_{sb} 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)
apply (clarsimp cong: memref.case-cong)
done
done

next
case (Read_{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)
apply (case-tac j)
apply (fastforce split: memref.splits simp add: in-read-tmps-conv dest: nth-mem)
apply (clarsimp cong: memref.case-cong)
done
apply clarsimp
subgoal for i j
  apply (erule-tac x=0 in allE)
  apply clarsimp
done

apply clarsimp
subgoal for i j
  apply (erule-tac x=Suc i in allE)
  apply clarsimp
done

next
case Prog_{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)
done
done

next
case Ghost sub
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

170
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:
  ⋀ i p is O R D d sb.
  [i < length ts; ts!i = (p, is, d, 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:
  ⋀ i p is O R D d sb.
  [i < length ts; ts!i = (p, is, d, 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:
  ⋀ i p is O R D d sb.
  [i < length ts; ts!i = (p, is, d, 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: ⋀ ys. distinct-load-tmps (xs @ ys) =
  (distinct-load-tmps xs ∧ distinct-load-tmps ys ∧
  load-tmps xs ∩ load-tmps ys = {})

171
apply (induct xs)
apply (auto split: instr.splits simp add: in-load-tmps-conv)
done

lemma read-tmps-append: read-tmps (xs@ys) = (read-tmps xs ∪ read-tmps ys)
  by (fastforce simp add: in-read-tmps-conv)

lemma load-tmps-append: load-tmps (xs@ys) = (load-tmps xs ∪ load-tmps ys)
  by (fastforce simp add: in-load-tmps-conv)

fun write-sops:: 'p store-buffer ⇒ sop set
  where
    write-sops [] = {}
    | write-sops (r#rs) =
      (case r of
        Write sb volatile a sop v - - - - ⇒ insert sop (write-sops rs)
        | - ⇒ write-sops rs)

lemma in-write-sops-conv:
  (sop ∈ write-sops xs) = (∃ volatile a v A L R W. Write sb volatile a sop v A L R W ∈ set xs)
  apply (induct xs)
  apply simp
  apply (auto split: memref.splits)
  apply force
  apply force
  done

lemma write-sops-mono: ∀ ys. set xs ⊆ set ys ⇒ write-sops xs ⊆ write-sops ys
  by (fastforce simp add: in-write-sops-conv)

lemma write-sops-append: write-sops (xs@ys) = write-sops xs ∪ write-sops ys
  by (force simp add: in-write-sops-conv)

fun store-sops:: instrs ⇒ sop set
  where
    store-sops [] = {}
    | store-sops (i#is) =
      (case i of
        Write volatile a sop - - - - ⇒ insert sop (store-sops is)
        | RMW a t sop cond ret A L R W ⇒ insert sop (store-sops is)
        | - ⇒ store-sops is)

lemma in-store-sops-conv:
  (sop ∈ store-sops xs) = (∃ volatile a A L R W. Write volatile a sop A L R W ∈ set xs)
  ∨
  (∃ a t 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 store-sops-mono: \( \forall ys. \text{set } xs \subseteq \text{set } ys \implies \text{store-sops } xs \subseteq \text{store-sops } ys \)
by (fastforce simp add: in-store-sops-conv)

lemma store-sops-append: store-sops (xs @ ys) = store-sops xs \cup \text{store-sops } ys
by (force simp add: in-store-sops-conv)

locale valid-write-sops =
fixes ts::\(\langle p, p \text{ store-buffer}, \text{bool, owns}, \text{rels} \rangle \) thread-config list
assumes valid-write-sops:
\( \forall i p \text{ is \ } ORD \theta \text{ sb}. [\ [i < \text{length } ts; ts!i = (p, \text{is, } \theta, \text{sb, } D, O, R)] \] \implies \forall \text{sop } \in \text{write-sops } \text{sb}. \text{valid-sop } \text{sop} \)

locale valid-store-sops =
fixes ts::\(\langle p, p \text{ store-buffer}, \text{bool, owns}, \text{rels} \rangle \) thread-config list
assumes valid-store-sops:
\( \forall i \text{ is } ORD \theta \text{ sb}. [\ [i < \text{length } ts; ts!i = (p, \text{is, } \theta, \text{sb, } D, O, R)] \] \implies \forall \text{sop } \in \text{store-sops } \text{is}. \text{valid-sop } \text{sop} \)

locale valid-sops = valid-write-sops + 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.

fun reads-consistent:: bool \Rightarrow \text{owns} \Rightarrow \text{memory} \Rightarrow \langle p \text{ store-buffer} \Rightarrow \text{bool} \rangle
where
reads-consistent pending-write \text{O m } [] = \text{True}
| reads-consistent pending-write \text{O m } (r#rs) =
  (case r of
    \text{Read}_{sb} \text{volatile a t v} \Rightarrow (\neg \text{volatile } \rightarrow (\text{pending-write } \vee a \in \text{O}) \rightarrow v = m a) \land
    \text{reads-consistent pending-write } \text{O m } \text{rs}
    | \text{Write}_{sb} \text{volatile a sop v A L R W} \Rightarrow
      (\text{if volatile then }
        \text{outstanding-refs is-volatile-Read}_{sb} \text{rs} = \{\} \land
        \text{reads-consistent True } (\text{O } \cup \text{A } \setminus \text{R}) (m(a := v)) \text{rs}
      \text{else reads-consistent pending-write } \text{O } (m(a := v)) \text{rs})
    | \text{Ghost}_{sb} \text{A L R W} \Rightarrow \text{reads-consistent pending-write } (\text{O } \cup \text{A } \setminus \text{R}) \text{m } \text{rs}
    | - \Rightarrow \text{reads-consistent pending-write } \text{O } \text{m } \text{rs}
    )

fun volatile-reads-consistent:: \text{memory} \Rightarrow \langle p \text{ store-buffer} \Rightarrow \text{bool} \rangle
where
volatile-reads-consistent m [] = \text{True}
| volatile-reads-consistent m (r#rs) =
  (case r of
    \text{Read}_{sb} \text{volatile a t v} \Rightarrow (\text{volatile } \rightarrow v = m a) \land \text{volatile-reads-consistent m } \text{rs}
    | \text{Write}_{sb} \text{volatile a sop v A L R W} \Rightarrow \text{volatile-reads-consistent } (m(a := v)) \text{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:
\[ \bigwedge O m. \text{reads-consistent True } O m sb \Rightarrow \text{reads-consistent False } O m sb \]
apply (induct sb)
apply simp
subgoal for a sb O m
  by (case-tac a) auto
done

lemma reads-consistent-owns-antimono:
\[ \bigwedge O O' \text{ pending-write } m. O \subseteq O' \Rightarrow \text{reads-consistent pending-write } O' m sb \Rightarrow \text{reads-consistent pending-write } O m sb \]
apply (induct sb)
apply simp
subgoal for a sb O O' pending-write m
apply (case-tac a)
apply (clarsimp 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 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
done

lemma acquired-reads-mono': x ∈ acquired-reads b xs A \Rightarrow acquired-reads b xs B = {}
⇒ A ⊆ B ⇒ False
apply (drule acquired-reads-mono-in [where B=B])
apply auto
done
lemma reads-consistent-append:
\( \forall m. \text{pending-write } O \cdot \text{reads-consistent pending-write } O \ m \ (xs@ys) = \)
\( (\text{reads-consistent pending-write } O \ m \ xs \land \text{reads-consistent } (\text{pending-write } \lor \text{outstanding-refs is-volatile-Write}_{sb} \ xs \neq \{\}) ) \)
\( (\text{acquired } \text{True } xs \ O) \ (\text{flush } xs \ m) \ ys \land \)
\( (\text{outstanding-refs is-volatile-Write}_{sb} \ xs \neq \{\}) \)
\( \rightarrow \text{outstanding-refs is-volatile-Read}_{sb} \ ys = \{\} \) \)
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
\( \) \text{dest: } \text{acquired-reads-mono-in acquired-pending-write-mono-in acquired-reads-mono'}
\( \text{acquired-mono-in} \) )
done
done

lemma reads-consistent-mem-eq-on-non-volatile-reads:
\( \text{assumes mem-eq: } \forall a \in A. \ m' \ a = m \ a \)
\( \text{assumes subset: } \text{outstanding-refs } (\text{is-non-volatile-Read}_{sb}) \ sb \subseteq A \)
\( \quad \text{— We could be even more restrictive here, only the non volatile reads that are not} \)
\( \quad \text{buffered in } sb \text{ have to be the same.} \)
\( \text{assumes consis-m: } \text{reads-consistent pending-write } O \ m \ sb \)
\( \text{shows } \text{reads-consistent pending-write } O \ m' \ sb \)
using mem-eq subset consis-m
proof (induct sb arbitrary: \( m' \) \(:\ m \) pending-write \( O \) )
\text{Nil thus } ?\text{case by simp}
next
\text{case } (\text{Cons } r \ sb)
\text{note mem-eq = } \forall a \in A. \ m' \ a = m \ a
\text{note subset = } \{\text{outstanding-refs } (\text{is-non-volatile-Read}_{sb}) \ (r#sb) \subseteq A\}
\text{note consis-m = } \{\text{reads-consistent pending-write } O \ m \ (r#sb)\}
from subset have subset': \(\) \text{outstanding-refs is-non-volatile-Read}_{sb} \ sb \subseteq A
\text{by (auto simp add: Write}_{sb} )
show ?\text{case}
proof (cases \( r \) )
\text{case } (\text{Write}_{sb} \text{ volatile } a \ \text{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 ?\text{thesis}
proof (cases \text{volatile} )
\text{case True}
\text{from consis-m obtain}\n\text{\text{consis'}: } \text{reads-consistent } \text{True } (O \cup A' - R) \ (m(a := v)) \ sb \ \text{and}
\text{\text{no-volatile-Read}_{sb}': } \text{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′:
    \( \forall a' \in A. \; m'(a := v) = m(a) \)
  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 sb True)
    from Cons.hyps [OF mem-eq subset consis]
    show ?thesis
    by (simp add: Read sb True)
  next
    case False
    from consis-m obtain consis′: reads-consistent pending-write O m sb and v: (pending-write ∨ a ∈ O) \( \rightarrow \) v = m a
    by (simp add: Read sb False)
    from mem-eq subset Read sb 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 sb False)
    qed
  next
    case Prog sb 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_{sb}) 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 = \( \forall a \in A. \, m' a = m a \)
  note subset = \langle outstanding-refs (is-volatile-Read_{sb}) (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_{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': volatile-reads-consistent (m(a := v)) sb
    by (simp add: Write_{sb} True)
      from Cons.hyps [OF mem-eq' subset' consis']
    have volatile-reads-consistent (m'(a := v)) sb.
    then
    show ?thesis
    by (simp add: Write_{sb} True)
next
    case False
    from consis-m obtain consis': volatile-reads-consistent (m(a := v)) sb
    by (simp add: Write_{sb} 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\textsubscript{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\textsubscript{m} obtain consis': volatile-reads-consistent m sb
by (simp add: Read\textsubscript{sb} False)
from Cons.hyps [OF mem-eq' subset' consis']
show ?thesis
by (simp add: Read\textsubscript{sb} False)
next
case True
from consis\textsubscript{m} obtain consis': volatile-reads-consistent m sb and v = m a
by (simp add: Read\textsubscript{sb} True)
from mem-eq subset Read\textsubscript{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\textsubscript{sb} True)
qed
next
case Prog\textsubscript{sb} with Cons show ?thesis by auto
next
case Ghost\textsubscript{sb} with Cons show ?thesis by auto
qed
qed
locale valid-reads =
fixes m::memory and ts::(('p,'p store-buffer,bool,owns,rels) thread-config list
assumes valid-reads:
\[ i < \text{length ts}; ts!i = (p,is,\theta,sb,D,O,R) \] \implies
reads-consistent False O m sb

lemma valid-reads-Cons: valid-reads m (t#ts) =
(let (\_,\_,\_,sb,\_,\_,\_,\_) = t in reads-consistent False O m sb \land valid-reads m ts)
apply (auto simp add: valid-reads-def)
subgoal for p' is' \theta' sb' D' O' R' \ i p is \emptyset sb D O R
apply (case-tac i)
apply auto
done
done

Read\textsubscript{sb}s 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 Ṵ p [] = True
  | history-consistent Ṵ p (r#rs) =
    (case r of
      Readšb vol a t v ⇒
        (case Ṵ t of Some v′ ⇒ v=v′ ∧ history-consistent Ṵ p rs | - ⇒ False)
    | Writešb vol a (D,f) v - - - - ⇒
        D ⊆ dom Ṵ ∧ f Ṵ = v ∧ D ∩ read-tmps rs = {} ∧ history-consistent Ṵ p rs
    | Progšb p_1 p_2 is ⇒ p_1=p ∧
        Ṵ(j)(dom Ṵ − read-tmps rs)⊢ p_1 →_p (p_2,is) ∧
        history-consistent Ṵ p_2 rs
    | - ⇒ history-consistent Ṵ p rs)

end

fun hd-prog:: 'p ⇒ 'p store-buffer ⇒ 'p

where

  hd-prog p [] = p
  | hd-prog p (i#is) = (case i of
    Progšb 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
    Progšb - p'- - ⇒ last-prog p'
  | - ⇒ last-prog p is)

locale valid-history = program +

constraints

  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 Ṵ sb.

  [i < length ts; ts!i = (p,is,Ṵ,šb,D,O,R) ] −−>
  program.history-consistent program-step Ṵ (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
    | RMW a t (D,f) cond ret - - - - ⇒ D ⊆ T ∧ D ∩ load-tmps is = {} ∧
      data-dependency-consistent-instrs (insert t T) is
    | Read - - t ⇒ data-dependency-consistent-instrs (insert t T) is
    | - ⇒ data-dependency-consistent-instrs T is)

lemma data-dependency-consistent-mono:
\[ \forall T, T'. \ \text{[data-dependency-consistent-instrs} \ T \ \text{is;} \ T \subseteq T' \text{]} \implies \text{data-dependency-consistent-instrs} \ T' \text{is} \]

apply (induct is)
apply clarsimp
subgoal for a is T T'
apply (case-tac a)
apply clarsimp
subgoal for volatile a' t
apply (drule-tac a=t in insert-mono)
apply clarsimp
done
apply fastforce
apply clarsimp
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
apply fastforce
done
done
done

lemma data-dependency-consistent-instrs-append:
\[ \forall ys T. \ \text{data-dependency-consistent-instrs} \ T \ (xs@ys) = \]
(data-dependency-consistent-instrs \ T \ xs \wedge
\text{data-dependency-consistent-instrs} \ (T \cup \text{load-tmps} \ \xs) \ ys \wedge
\text{load-tmps} \ \ys \cap \bigcup (\text{fst} \ \cdot \ \text{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:
\[ \forall i \ p \ \text{is} \ \mathcal{D} \ \emptyset \ \text{sb}. \]
\[ \ [i < \text{length} \ \text{ts}; \ \text{ts}[i] = \ (\text{p}, \text{is},\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R}) \ ] \implies \text{data-dependency-consistent-instrs} \ (\text{dom} \ \emptyset) \text{is} \]
assumes load-tmps-write-tmps-distinct:
\[ \forall i \ p \ \text{is} \ \mathcal{D} \ \emptyset \ \text{sb}. \]
\[ \ [i < \text{length} \ \text{ts}; \ \text{ts}[i] = \ (\text{p}, \text{is},\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R}) \ ] \implies \text{load-tmps} \ \text{is} \cap \bigcup (\text{fst} \ \cdot \ \text{write-sops} \ \text{sb}) = \{\}

locale load-tmps-fresh =
fixes ts::('p, 'p store-buffer,bool,owns,rels) thread-config list
assumes load-tmps-fresh:
\[ \forall i \ p \ \text{is} \ \mathcal{D} \ \emptyset \ \text{sb}. \]
\[ \ [i < \text{length} \ \text{ts}; \ \text{ts}[i] = \ (\text{p}, \text{is},\emptyset,\text{sb},\mathcal{D},\mathcal{O},\mathcal{R}) \ ] \implies \text{load-tmps} \ \text{is} \cap \text{dom} \ \emptyset = \{\}

180
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 - - - ⇒ acquired-by-instrs is A
      | Write volatile - - A’ L R W ⇒ acquired-by-instrs is (if volatile then (A ∪ 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 ∪ 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 ∧ ¬ volatile ∧ a ∈ A
            then insert a (acquired-loads pending-write is A) else acquired-loads pending-write is A)
      | Write volatile - - A’ L R W ⇒ (if volatile then acquired-loads True is (if pending-write then (A ∪ 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 ∪ A’ − R))

lemma acquired-by-instrs-mono:
    ∀ A B. A ⊆ B ⇒ acquired-by-instrs is A ⊆ acquired-by-instrs is B

apply (induct is)
apply simp
subgoal for a is A B
apply (case-tac a)
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

181
done

**lemma** acquired-by-instrs-mono-in:

**assumes** x-in: $x \in \text{acquired-by-instrs is } A$

**assumes** sub: $A \subseteq B$

**shows** $x \in \text{acquired-by-instrs is } B$

**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:

$I \in \text{length ts; ts!i = (p,is,}\varnothing,\text{sb,}O,\text{R)} \implies (\text{outstanding-refs is-volatile-Write}_{sb} \text{ sb} = \{\})$

**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:

$I \forall sb_1 sb_2 . \text{sb=}@sb_2 \implies \exists \text{ instrs sb}_2 \oplus is_{sb} = is \oplus \text{ prog-instrs sb}_2$

**lemma** causal-program-history-empty [simp]: causal-program-history is []

**by** (rule causal-program-history.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:

$I \forall i p is \varnothing R \text{ }D \varnothing \text{ sb}.

[I < \text{length ts; tsli = (p,is,}\varnothing,\text{sb,}D,\text{R)} \implies ]$

182
causal-program-history is sb

assumes valid-last-prog:
\[ \forall i \ p \in \text{ORD} \ \theta \ sb. \ [i < \text{length ts}; ts!i = (p, is, \theta, sb, D, O, R)] \implies \text{last-prog } p \ sb = p \]

lemma (in valid-program-history) valid-program-history-nth-update:
\[ [i < \text{length ts}; \text{causal-program-history is sb}; \text{last-prog } p \ sb = p] \implies \text{valid-program-history (ts } [i := (p, is, \theta, sb, D, O, R)]) \]
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:
\[ \forall i \ p \in \text{ORD} \ \theta \ sb. \ [i < \text{length ts}; ts!i = (p, is, \theta, sb, D, O, R)] \implies \text{outstanding-non-volatile-refs-owned-or-read-only } S (ts[i := (p, is', \theta', sb, D', O, R)]) \]
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:
\[ \forall i \ is \ O R D \ \theta \ sb. \ [i < \text{length ts}; \text{non-volatile-owned-or-read-only } False \ S \ sb] \implies \text{outstanding-non-volatile-refs-owned-or-read-only } S (ts[i := (p, is, \theta, sb, D, O, R)]) \]
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:
\[ \forall i \ p \in \text{ORD} \ \theta \ sb. \ [i < \text{length ts}; ts!i = (p, is, \theta, sb, D, O, R)] \implies \text{outstanding-volatile-writes-unowned-by-others } (ts[i := (p, is', \theta', sb, D', O, R)]) \]
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:
\[ \forall i \ p \in \text{ORD} \ \theta \ sb. \ [i < \text{length ts}; ts!i = (p, is, \theta, sb, D, O, R)] \implies \text{read-only-unowned } (ts[i := (p, is', \theta', sb, D', O, R)]) \]
by (unfold-locales)
  (auto dest: read-only-unowned simp add: nth-list-update split: if-split-asm)
lemma \textit{Write}_{sb}-in-outstanding-refs:
  Write_{sb} \text{ True a sop v A L R W \in set xs} \implies a \in \text{outstanding-refs is-volatile-Write}_{sb} \ x s
  by (induct xs) (auto split:memref.splits)

lemma (in outstanding-volatile-writes-unowned-by-others)
outstanding-volatile-writes-unowned-by-others-store-buffer:
\exists i \ p \is (O \ R \ D \ \varnothing \ sb).
\forall i < \text{length ts}; ts!i = (p,\text{is},\varnothing,\text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R});
\exists outstanding-refs is-volatile-Write_{sb} sb' \subseteq outstanding-refs is-volatile-Write_{sb} sb;
\exists all-acquired sb' \subseteq all-acquired sb\implies
\forall outstanding-volatile-writes-unowned-by-others (ts[i := (p',\text{is}',\varnothing',\text{sb}', \mathcal{D}', \mathcal{O}, \mathcal{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:
\exists i \ p \is (O \ R \ D \ \varnothing \ sb).
\forall i < \text{length ts}; ts!i = (p,\text{is},\varnothing,\text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R});
\exists all-acquired sb' \subseteq all-acquired sb\implies
\forall ownership-distinct (ts[i := (p',\text{is}',\varnothing',\text{sb}', \mathcal{D}', \mathcal{O}, \mathcal{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:
\exists i \ p \is (O \ R \ D \ xs \ sb).
\forall i < \text{length ts}; ts!i = (p,\text{is},\varnothing,\text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R});
\forall j < \text{length ts}. i \neq j \implies (let (p_j,\text{is}_j,\varnothing_j,\text{sb}_j, \mathcal{D}_j, \mathcal{O}_j, \mathcal{R}_j) = ts!j)
in (O' \cup \text{all-acquired sb}') \cap (O_j \cup \text{all-acquired sb}_j) = \{\} \implies
ownership-distinct (ts[i := (p',\text{is}',\varnothing',\text{sb}', \mathcal{D}', \mathcal{O}', \mathcal{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)
done

lemma (in valid-write-sops) valid-write-sops-nth-update:
\forall i < \text{length ts}; \forall sop \in \text{write-sops sb}. valid-sop sop \implies
valid-write-sops (ts[i := (p,\text{is},\text{xs},\text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{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
valid-store-sops (ts[i := (p,is,xs,\theta,db,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
valid-sops (ts[i := (p,is,xs,\theta,db,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:
[\[i < \text{length } ts; \text{data-dependency-consistent-instrs (dom } \theta \text{) is};
\text{load-tmps is } \cap \bigcup (\text{fst } ' \text{write-sops sb}) = \{\} \implies
\text{valid-data-dependency (ts[i := (p,is,\theta,db,O,R)])}\]
by (unfold valid-data-dependency-def)
(force dest: data-dependency-consistent-instrs load-tmps-write-tmps-distinct
simp add: nth-list-update split: if-split-asm)

lemma (in enough-flushs) enough-flushs-nth-update:
[\[i < \text{length } ts;
\neg \exists \text{D} \rightarrow (\text{outstanding-refs is-volatile-Write}_{sb} \text{ sb} = \{\})\]
\implies
enough-flushs (ts[i := (p,is,\theta,db,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:
[\[i < \text{length } ts; \text{non-volatile-writes-unshared } S \text{ sb}] \implies
\text{outstanding-non-volatile-writes-unshared } S \text{ (ts[i := (p,is,xs,\theta,db,O,R)])}\]
by (unfold-locales)
(auto dest: outstanding-non-volatile-writes-unshared
simp add: nth-list-update split: if-split-asm)

lemma (in sharing-consis)
sharing-consis-nth-update:
[\[i < \text{length } ts; \text{sharing-consistent } S \text{ sb}] \implies
\text{sharing-consis } S \text{ (ts[i := (p,is,xs,\theta,db,O,R)])}\]
by (unfold-locales)
lemma (in no-outstanding-write-to-read-only-memory)
no-outstanding-write-to-read-only-memory-nth-update:
\[ [i < \text{length } ts; \text{no-write-to-read-only-memory } S \text{ sb}] \implies \text{no-outstanding-write-to-read-only-memory } S (ts[i := (p,is,xs,sb,D,O,R)]) \]
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 \( \implies \exists i. \ i < \text{length } xs \land a \in f (xs!i) \)
by (auto simp add: in-set-conv-nth)

lemma in-Inter-image-nth-conv: \( a \in \bigcap f \) set xs = (\( \forall i < \text{length } xs. a \in f (xs!i) \))
by (force simp add: in-set-conv-nth)

lemma release-ownership-nth-update:
assumes R-subset: \( R \subseteq O \)
shows \( \forall i. [i < \text{length } ts; ts!i = (p,is,xs,sb,D,O,R)]; \)
ownership-distinct ts\]
\( \implies \bigcup ((\lambda(\_\_\_\_\_\_\_,O,\_\_\_\_\_\_\_), O) \ i \ set (ts[i:=(p,is',xs',sb',D',O - R,\_\_\_\_\_\_\_)))) = (\bigcup ((\lambda(\_\_\_\_\_\_\_,O,\_\_\_\_\_\_\_), O) \ i \ set ts)) - R \)
proof (induct ts)
case Nil thus ?case by simp
next
case (Cons t ts)
\text{note} i-bound = \( \langle i < \text{length } (t \# ts) \rangle \)
\text{note} ith = \( \langle (t \# ts) ! i = (p,is,xs,sb,D,O,R) \rangle \)
\text{note} dist = \( \langle \text{ownership-distinct } (t\#ts) \rangle \)
then interpret \( \text{ownership-distinct } t\#ts. \)
from dist
have dist': \( \text{ownership-distinct } ts. \)
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) \ i \ set ts)) = \{\}
proof -
\{
fix x
assume x-R: \( x \in R \)
assume x-ls: \( x \in (\bigcup ((\lambda(\_\_\_\_\_\_\_,O,\_\_\_\_\_\_\_), O) \ i \ set ts)) \)
then obtain \( j \ p_j \ is_j \ O_j \ D_j \ xs_j \ sb_j \) where
\}
j-bound: \( j < \text{length ts} \) and
jth: \( \text{ts}!j = (p_j, i_s_j, x_s_j, s_b_j, D_j, O_j, R_j) \) and
x-in: \( x \in O_j \)
by (fastforce simp add: in-set-conv-nth )
from j-bound jth 0
have \( (O \cup \text{all-acquired sb}) \cap (O_j \cup \text{all-acquired sb}_j) = {} \)
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 blast
qed
then
show ?thesis
by (auto simp add: 0 t)
next
case (Suc n)
obtain \( p_l, i_s_l, O_l, R_l, D_l, x_s_l, s_b_l \) where \( t = (p_l, i_s_l, x_s_l, s_b_l, D_l, O_l, R_l) \)
by (cases t)

have n-bound: \( n < \text{length ts} \)
using i-bound by (simp add: Suc)
have nth: \( \text{ts}!n = (p_i, i_s_i, x_s_i, s_b_i, D_i, O_i, R_i) \)
using ith by (simp add: Suc)

have \( R \cap (O_i \cup \text{all-acquired sb}_i) = {} \)
proof −
{ fix x 
assume x-R: \( x \in R \)
assume x-owns\_i: \( x \in (O_i \cup \text{all-acquired sb}_i) \)
from t
have \( (O \cup \text{all-acquired sb}) \cap (O_i \cup \text{all-acquired sb}_i) = {} \)
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
with Cons.hyps [OF n-bound nth dist']
show ?thesis
  by (auto simp add: Suc t)
qed

lemma acquire-ownership-nth-update:
  shows \( \forall i. \ [i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, \mathcal{R})] \)
    \( \implies \bigcup \ ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). \ O)^\cdot \text{ set } \text{ts}[i:==(p', is', xs', sb', D', O \cup A, \mathcal{R}')]) \)
  = \((\bigcup \ ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). \ O)^\cdot \text{ set } \text{ts})) \cup A \)
proof (induct ts)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  note i-bound = \( \langle i < \text{length } (t \# ts) \rangle \)
  note ith = \( \langle (t \# ts)!i = (p, is, xs, sb, D, O, R) \rangle \)
  show ?thesis
    proof (cases i)
      case 0
      from ith 0 have t: \( t = (p, is, xs, sb, D, O, R) \)
        by simp
      show ?thesis
        by (auto simp add: 0 t)
    next
      case (Suc n)
      obtain p_l is_l R_l D_l xs_l sb_l where t: \( t = (p_l, is_l, xs_l, sb_l, D_l, O_l, R_l) \)
        by (cases t)
      have n-bound: \( n < \text{length } ts \)
        using i-bound by (simp add: Suc)
      have nth: \( ts!n = (p, is, xs, sb, D, O, R) \)
        using ith by (simp add: Suc)
      from Cons.hyps [OF n-bound nth]
      show ?thesis
        by (auto simp add: Suc t)
    qed
qed

lemma acquire-release-ownership-nth-update:
  assumes R-subset: \( R \subseteq O \)
  shows \( \forall i. \ [i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, \mathcal{R})] \)
    \( \implies \bigcup \ ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). \ O)^\cdot \text{ set } \text{ts}[i:==(p', is', xs', sb', D', O \cup A - R, \mathcal{R}')]) \)
  = \((\bigcup \ ((\lambda(\cdot, \cdot, \cdot, \cdot, O, \cdot). \ O)^\cdot \text{ set } \text{ts})) \cup A - R \)
proof (induct ts)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  note i-bound = \( \langle i < \text{length } (t \# ts) \rangle \)
  note ith = \( \langle (t \# ts)!i = (p, is, xs, sb, D, O, \mathcal{R}) \rangle \)
  note dist = \( \langle \text{ownership-distinct } (t \# ts) \rangle \)
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 ∩ (⋃ (λ(·,·,·,·,O,·). O) ^ i 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 is_j O_j D_j x_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

next
  case (Suc n)
  obtain p_l is_l O_l D_l x_l sb_l where t: t = (p_l,is_l,xs_l,sb_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,is,xs,sb,D,O,R)
    using ith by (simp add: Suc)

  have R ∩ (O_l ∪ all-acquired sb_l) = {}
  proof
    
    189
fix x
assume x-R: x ∈ R
assume x-ownsₙ: x ∈ (𝒪ₙ ∪ all-acquired sbₙ)
from t
have (𝒪 ∪ all-acquired sb) ∩ (𝒪ₙ ∪ all-acquired sbₙ) = {}
apply –
apply (rule ownership-distinct [OF i-bound - - ith, of 0])
apply (auto simp add: Suc)
done
with x-ownsₙ 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
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)])
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 < length ts; load-tmps is ∩ read-tmps sb = {}] ⇒ load-tmps-read-tmps-distinct (ts[i := (p,is,xs,sb,D,O,R)])
lemmaw (in load-tmps-fresh) load-tmps-fresh-nth-update:
\[ [i < \text{length ts};
  \text{load-tmps is } \cap \text{dom } \emptyset = \{\}] \implies \text{load-tmps-fresh (ts[i := (p, is, sb)}] \]
by (unfold-locales)
(fastforce dest: load-tmps-fresh
  simp add: nth-list-update split: if-split-asm)

fun flush-all-until-volatile-write::
(\'p store-buffer,\'dirty,\'owns,\'rels) thread-config list \Rightarrow memory \Rightarrow memory
where
flush-all-until-volatile-write [] m = m
| flush-all-until-volatile-write ((\cdot, \cdot, \cdot, sb, \cdot, \cdot)\#ts) m =
  flush-all-until-volatile-write ts (flush (takeWhile (\neg \circ \text{is-volatile-Write}) sb) m)

fun share-all-until-volatile-write::
(\'p store-buffer,\'dirty,\'owns,\'rels) thread-config list \Rightarrow shared \Rightarrow shared
where
share-all-until-volatile-write [] S = S
| share-all-until-volatile-write ((\cdot, \cdot, \cdot, sb, \cdot, \cdot)\#ts) S =
  share-all-until-volatile-write ts (share (takeWhile (\neg \circ \text{is-volatile-Write}) sb) S)

lemmatakeWhile-dropWhile-real-prefix:
\[ [x \in \text{set xs}; \neg P x] \implies \exists y \text{ys. } \text{xstakeWhile P xs @ y#ys} \land \neg P y \land \text{dropWhile P xs = y#ys} \]
by (induct xs) auto

lemmabuffered-val-witness: buffered-val sb a = Some v \implies
\exists \text{volatile sop A L R W. 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
done

lemmashellot flush-append-Read_{sb}:
\[ \forall m. (\text{flush (takeWhile (\neg \circ \text{is-volatile-Write}_{sb}) (sb @ [Read}_{sb} \text{volatile a t v]))) m) \]
= flush (takeWhile (\neg \circ \text{is-volatile-Write}_{sb}) sb) m

191
by (induct sb) (auto split: memref.splits)

**lemma** flush-append-write:
\[ \forall m. (\text{flush } (sb @ [\text{Write}_{sb} \text{ volatile } a \text{ sop } v \ A \ L \ R \ W]) m) = (\text{flush } sb m) (a:=v) \]
by (induct sb) (auto split: memref.splits)

**lemma** flush-append-Prog_{sb}:
\[ \forall m. (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) (sb @ [\text{Prog}_{sb} p_1 p_2 \text{ mis}])) m) = (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb) m) \]
by (induct sb) (auto split: memref.splits)

**lemma** flush-append-Ghost_{sb}:
\[ \forall m. (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) (sb @ [\text{Ghost}_{sb} A \ L \ R \ W])) m) = (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb) m) \]
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-Read_{sb}:
\[ \forall S. (\text{share } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) (sb @ [\text{Read}_{sb} \text{ volatile } a \ t \ v])) S) = \text{share } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb) S \]
by (induct sb) (auto split: memref.splits)

**lemma** share-append-Write_{sb}:
\[ \forall S. (\text{share } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) (sb @ [\text{Write}_{sb} \text{ volatile } a \text{ sop } v \ A \ L \ R \ W])) S) = \text{share } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb) S \]
by (induct sb) (auto split: memref.splits)

**lemma** share-append-Prog_{sb}:
\[ \forall S. (\text{share } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) (sb @ [\text{Prog}_{sb} p_1 p_2 \text{ mis}])) S) = \text{share } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb) S \]
by (induct sb) (auto split: memref.splits)

**lemma** in-acquired-no-pending-write-outstanding-write:
\[ a \in \text{acquired False } sb A \Longrightarrow \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\} \]
apply (induct sb)
apply (auto split: memref.splits)
done

**lemma** flush-buffered-val-conv:
\[ \forall m. \text{flush } sb m a = (\text{case } \text{buffered-val } sb a \text{ of } \text{None } \Rightarrow m a \mid \text{Some } v \Rightarrow v) \]
by (induct sb) (auto split: memref.splits option.splits)

**lemma** reads-consistent-unbuffered-snoc:
\[ \forall m. \text{buffered-val } sb a = \text{None } \Longrightarrow m a = v \Longrightarrow \text{reads-consistent pending-write } O m sb \]
\[ \Longrightarrow \]

192
volatile $$\rightarrow$$
outstanding-refs is-volatile-Write sb = {}

$$\Rightarrow$$ reads-consistent pending-write $$\mathcal{O}$$ m (sb @ [Read sb volatile a t v])
by (simp add: reads-consistent-append flush-buffered-val-conv)

**Lemma** reads-consistent-buffered-snoc:
$$\forall m. \text{buffered-val} \text{ sb a} = \text{Some v} \Rightarrow \text{reads-consistent pending-write } \mathcal{O} \text{ m sb} \Rightarrow$$
volatile $$\rightarrow$$ outstanding-refs is-volatile-Write sb = {}

$$\Rightarrow$$ reads-consistent pending-write $$\mathcal{O}$$ m (sb @ [Read sb volatile a t v])
by (simp add: reads-consistent-append flush-buffered-val-conv)

**Lemma** reads-consistent-snoc-Write sb:

$$\forall m. \text{reads-consistent pending-write } \mathcal{O} \text{ m sb} \Rightarrow$$
reads-consistent pending-write $$\mathcal{O}$$ m (sb @ [Write sb volatile a sop v A L R W])
by (simp add: reads-consistent-append)

**Lemma** reads-consistent-snoc-Prog sb:

$$\forall m. \text{reads-consistent pending-write } \mathcal{O} \text{ m sb} \Rightarrow$$
reads-consistent pending-write $$\mathcal{O}$$ m (sb @ [Prog sb p1 p2 mis])
by (simp add: reads-consistent-append)

**Lemma** reads-consistent-snoc-Ghost sb:

$$\forall m. \text{reads-consistent pending-write } \mathcal{O} \text{ m sb} \Rightarrow$$
reads-consistent pending-write $$\mathcal{O}$$ m (sb @ [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 $$\forall m_i. \quad [i < \text{length ls}; \text{ls}!i=(p, \text{Read volatile a t is,} \text{sb,} \mathcal{D}, \mathcal{O}, \mathcal{R})]$$
$$\quad$$
$$\Rightarrow$$
flush-all-until-volatile-write
$$(\text{ls}[i := (p, \text{is} , \vartheta(t\rightarrow v)), \text{sb} @ [\text{Read sb volatile a t v}]\mathcal{D}', \mathcal{O}', \mathcal{R}']) \text{ m} =$$
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 = \((l#ls)!i = (p, \text{Read volatile a t is,} \text{sb,} \mathcal{D}, \mathcal{O}, \mathcal{R})\)
show ?case
proof (cases i)
  case 0
  from ith 0 have l: l = (p, 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 O l D l \theta l sb l where l: l = (p, is, \theta, sb, D, O, R)
    by (cases l)
  from i-bound ith have flush-all-until-volatile-write
    (ls[n := (p, is, \theta(t \mapsto v), sb @ [Read sb volatile a t v], D', O', R']) ])
    (flush (takeWhile (Not o is-volatile-Write_{sb}) sb) m) =
    flush-all-until-volatile-write ls (flush (takeWhile (Not o is-volatile-Write_{sb}) sb) 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:
\[ \forall i m. [i < length ts; ts!i = (p, is, \theta, sb, D, O, R)] \]
\[ \Rightarrow flush-all-until-volatile-write (ts[i := (p', is', \theta', sb@[Ghost_{sb} A L R W], D', O', R')]]) m \]
\[ = flush-all-until-volatile-write ts m \]
proof (induct ts)
  case Nil thus ?case
    by simp
next
  case (Cons l ts)
  note i-bound = \[ i < length (l#ts) \]
  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 is l O l D l \theta l sb l where l: l = (p, is, \theta, sb, D, O, R)
    by (cases l)
from i-bound ith

have flush-all-until-volatile-write
  (ts |n := (p', is', θ', sb@[Ghost sb A L R W], D', O', R'))
  (flush (takeWhile (Not o is-volatile-Write sb) sb) m) =
  flush-all-until-volatile-write ts
  (flush (takeWhile (Not o is-volatile-Write sb) sb) 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: ∀ a m. a /∈ G → g m a = m a
assumes g-independent: ∀ a m. a ∈ G → g (f m) a = g m a
assumes f-unchanged: ∀ a m. a /∈ F → f m a = m a
assumes f-independent: ∀ a m. a ∈ F → f (g m) a = f m a
assumes disj: G ∩ F = {}
shows f (g m) = g (f m)
proof
  fix a
  show f (g m) a = g (f m) a
  proof (cases a ∈ G)
    case True
    with disj have a-notin-F: a /∈ F
    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 ... = g (f m) a by simp
    finally show ?thesis .
  next
    case False
    note a-notin-G = this
    show ?thesis
    proof (cases a ∈ 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 ... = 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 ... = m a.
  also
  from f-unchanged [rule-format, OF False]
  have ... = f m a by simp
  also
  from g-unchanged [rule-format, OF a-notin-G]
  have ... = g (f m) a
by simp
  finally show ?thesis.
  qed
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: \( \forall m.\ a \notin outstanding-refs is-Write sb \Rightarrow flush sb m a = m a \)
proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons r sb)
  note a-notin = \( \langle a \notin outstanding-refs is-Write sb \rangle \)
  show ?case
proof (cases r)
  case (Write sb volatile a' sop v)
  from a-notin obtain neq-a-a': \( a \neq a' \) and a-notin': \( a \notin outstanding-refs is-Write sb \)
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

196
done

next
case (Read$_{sb}$ volatile a$'$ t v)
  from a-notin obtain a-notin$: a \notin$ 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:
\[ \forall m m' a. a \in$ outstanding-refs is-Write$_{sb}$ sb \implies$ flush sb m' a = flush sb m a \]
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 flush sb (m'(a$'$:= v$'$)) a' = flush sb (m(a$'$:= v$'$)) a'
proof (cases a$'$ $\in$ 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
next
case Ghost$_{sb}$ thus $?thesis$ using Cons
  by auto
qed
qed
lemma flush-all-until-volatile-write-unchanged-addresses:
\[ \forall m. a \notin \bigcup ((\lambda (-,-,sb,-,-)). \text{outstanding-refs is-Write}_{sb}
\text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb))) \setminus \text{set ls}) \implies
\text{flush-all-until-volatile-write ls m a = m a} \]

proof (induct ls)
case Nil thus ?case by simp
next
case (Cons l ls)
obtain p is \(\mathcal{O} \mathcal{R} \mathcal{D} \mathcal{X} \mathcal{S} \mathcal{S} \mathcal{B}\) where l: l=(p, is, xs, sb, D, O, R) by (cases l)

note \(a \notin \bigcup ((\lambda (-,-,sb,-,-)). \text{outstanding-refs is-Write}_{sb}
\text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb))) \setminus \text{set (l#ls)}\)

then obtain
\(a-\text{notin-sb}: a \notin \text{outstanding-refs is-Write}_{sb}
\text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)}\) and
\(a-\text{notin-ls}: a \notin \bigcup ((\lambda (-,-,sb,-,-)). \text{outstanding-refs is-Write}_{sb}
\text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb))) \setminus \text{set ls}\)

by (auto simp add: l)

from Cons.hyps [OF a-notin-ls]

have flush-all-until-volatile-write ls (flush (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) m) a
\[=\]
(flush (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) m) a.

also

from flush-unchanged-addresses [OF a-notin-sb]

have (flush (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) 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 (Not \circ \text{is-volatile}) sb}
\[\implies\]
a \notin \text{outstanding-refs is-Write}_{sb} (takeWhile (Not \circ \text{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-lem':
a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} sb
\[\implies\]
a \notin \text{outstanding-refs is-Write}_{sb} (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)

apply (induct sb)

apply (auto simp add: is-Write_{sb}-def split: if-split-asm memref.splits)
done

198
lemma notin-outstanding-non-volatile-takeWhile-Un-lem':
a \notin \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ set ls})
\implies a \notin \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ 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: l=(p,is,xs,sb,D,O,R)
  by (cases l)

from Cons.prems
obtain
  a-notin-sb: a \notin \text{outstanding-refs is-volatile sb}
  and
  a-notin-ls: a \notin \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ set ls})
  by (force simp add: l simp del: o-apply)

from notin-outstanding-non-volatile-takeWhile-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 \notin \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ set ls})
shows flush-all-until-volatile-write ls m a = m a
using notin-outstanding-non-volatile-takeWhile-Un-lem' [OF notin]
by (auto intro: flush-all-until-volatile-write-unchanged-addresses)

lemma flush-all-until-volatile-write-mem-independent:
\forall m m'. a \in \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ set ls}) \implies
flush-all-until-volatile-write ls m a = flush-all-until-volatile-write ls m a
proof (induct ls)
  case Nil thus ?thesis
next
  case (Cons l ls)
  obtain p is O R D xs sb where l: l=(p,is,xs,sb,D,O,R)
  by (cases l)
  note a-in = \exists a \in \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ set (l\#ls)})

show ?thesis
  proof (cases a \in \bigcup ((\lambda(x,sb,r,R). \text{outstanding-refs is-Write}_{sb}
\text{takeWhile (Not o is-volatile-Write}_{sb} sb)) \iota \text{ 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
qed

lemma flush-all-until-volatile-write-buffered-val-conv:
assumes no-volatile-Write_{sb}: outstanding-refs is-volatile-Write_{sb} sb = {}
shows \(\forall m i. \ [i < \length{ls}; \ls!i = (p, is, xs, sb, D, O, R)]; \)
\(\forall j < \length{ls}. \ i \neq j \rightarrow\)
\(\begin{aligned}
& (let (-, -, -, sb_j, -, -) = \ls!j \\
in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile (Not ◦ is-volatile-Write}_{sb}) sb_j)) \end{aligned}\) \(\rightarrow\)
flush-all-until-volatile-write ls m a =
\(\begin{cases}
\text{flush-all-until-volatile-write-unchanged-addresses} & \text{if False} \\
\text{buffered-val sb a of None} & \Rightarrow m a | \text{Some v} \Rightarrow v
\end{cases}\)
proof (induct ls)
case Nil thus ?case
  by simp
next
case (Cons l ls)
note i-bound = \langle i < \length{(l#ls)} \rangle
note ith = \langle (l#ls)!i = (p, is, xs, sb, D, O, R) \rangle
note notin = \(\forall j < \length{(l#ls)}. \ i \neq j \rightarrow\)
\(\begin{aligned}
& (let (-, -, -, sb_j, -, -) = (l#ls)!j \\
in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile (Not ◦ is-volatile-Write}_{sb}) sb_j)) \end{aligned}\)
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_{sb} have take-all: takeWhile (Not ◦ is-volatile-Write_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
have a \notin \bigcup((\lambda(-, -, -, sb, -, -). \text{outstanding-refs is-Write}_{sb} (\text{takeWhile (Not ◦ is-volatile-Write}_{sb}) sb)) \ i \ \text{set ls}) (a \notin ?LS)
proof
assume a \in ?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) 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)

apply (clarsimp simp add: outstanding-refs-conv )

subgoal for x

apply (case-tac x)

apply clarsimp

apply (frule set-takeWhileD)

apply auto
done

done

withnotin [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

from flush-all-until-volatile-write-unchanged-addresses [OF this]

have flush-all-until-volatile-write ls (flush sb 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 = (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

= (case buffered-val sb a of None ⇒

(flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) m) a | Some v ⇒ v)

apply –

apply (case_tac x)

apply auto

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)

by (auto simp add: outstanding-refs-conv Suc )

then

have a ∉ outstanding-refs is-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb)

apply (clarsimp simp add: outstanding-refs-conv is-Write_{sb}-def split: memref.splits dest: set-takeWhileD)

apply (frule set-takeWhileD)

apply auto
from flush-unchanged-addresses [OF this]
have (flush (takeWhile (Not ◦ is-volatile-Write_{sb}) 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 ::
  (′p′p store-buffer,′dirty,′owns,′rels,′shared) global-config ⇒ (′p′p store-buffer,′dirty,′owns,′rels,′shared) global-config ⇒ bool
  (¬ ⇒_{sb} - [60,60] 100)
where
  sb-concurrent-step ≡
    computation.concurrent-step sb-memop-step store-buffer-step program-step (λp p′ is sb. sb)

term x ⇒_{sb} Y

abbreviation (in program) sb-concurrent-steps::
  (′p′p store-buffer,′dirty,′owns,′rels,′shared) global-config ⇒ (′p′p store-buffer,′dirty,′owns,′rels,′shared) global-config ⇒ bool
  (¬ ⇒_{sb}∗ - [60,60] 100)
where
  sb-concurrent-steps ≡ sb-concurrent-step^**

term x ⇒_{sb}∗ Y

abbreviation sbh-concurrent-step ::
  (′p′p store-buffer,bool,owns,rels,shared) global-config ⇒ (′p′p store-buffer,bool,owns,rels,shared) global-config ⇒ bool
  (¬ ⇒_{sbh} - [60,60] 100)
where
  sbh-concurrent-step ≡
    computation.concurrent-step sbh-memop-step flush-step program-step
    (λp p′ is sb. sb @ [Prog_{sb} p p′ is] )

term x ⇒_{sbh} Y

abbreviation sbh-concurrent-steps::
  (′p′p store-buffer,bool,owns,rels,shared) global-config ⇒ (′p′p store-buffer,bool,owns,rels,shared) global-config ⇒ bool
  (¬ ⇒_{sbh}∗ - [60,60] 100)
where

202
sbh-concurrent-steps ≡ sbh-concurrent-step∗∗

\[ \text{term } x \Rightarrow_{sb}^* Y \]
end

**Lemma** instrs-append-Read\textsubscript{sb}:

\[ \text{instrs (sb@[Read\textsubscript{sb} volatile a t v]) = instrs sb @ [Read volatile a t]} \]

by (induct sb) (auto split: memref.splits)

**Lemma** instrs-append-Write\textsubscript{sb}:

\[ \text{instrs (sb@[Write\textsubscript{sb} volatile a sop v A L R W]) = instrs sb @ [Write volatile a sop A L R W]} \]

by (induct sb) (auto split: memref.splits)

**Lemma** instrs-append-Ghost\textsubscript{sb}:

\[ \text{instrs (sb@[Ghost\textsubscript{sb} A L R W]) = instrs sb @ [Ghost A L R W]} \]

by (induct sb) (auto split: memref.splits)

**Lemma** prog-instrs-append-Ghost\textsubscript{sb}:

\[ \text{prog-instrs (sb@[Ghost\textsubscript{sb} A L R W]) = prog-instrs sb} \]

by (induct sb) (auto split: memref.splits)

**Lemma** prog-instrs-append-Read\textsubscript{sb}:

\[ \text{prog-instrs (sb@[Read\textsubscript{sb} volatile a t v]) = prog-instrs sb} \]

by (induct sb) (auto split: memref.splits)

**Lemma** prog-instrs-append-Write\textsubscript{sb}:

\[ \text{prog-instrs (sb@[Write\textsubscript{sb} volatile a sop v A L R W]) = prog-instrs sb} \]

by (induct sb) (auto split: memref.splits)

**Lemma** hd-prog-append-Read\textsubscript{sb}:

\[ \text{hd-prog p (sb@[Read\textsubscript{sb} volatile a t v]) = hd-prog p sb} \]

by (induct sb) (auto split: memref.splits)

**Lemma** hd-prog-append-Write\textsubscript{sb}:

\[ \text{hd-prog p (sb@[Write\textsubscript{sb} volatile a sop v A L R W]) = hd-prog p sb} \]

by (induct sb) (auto split: memref.splits)

**Lemma** flush-update-other: \( \forall m. \ a \notin \text{outstanding-refs (Not \circ is-volatile)} \) \( sb \implies \)

\[ \text{outstanding-refs (is-volatile-Write\textsubscript{sb}) sb = \{} \implies \]

\[ \text{flush sb (m(a:=v)) = (flush sb m)(a := v)} \]

by (induct sb)

(auto split: memref.splits if-split-asm simp add: fun-upd-twist)

**Lemma** flush-update-other': \( \forall m. \ a \notin \text{outstanding-refs (is-non-volatile-Write\textsubscript{sb})} \) \( sb \implies \)

\[ \text{outstanding-refs (is-volatile-Write\textsubscript{sb}) sb = \{} \implies \]

\[ \text{flush sb (m(a:=v)) = (flush sb m)(a := v)} \]

by (induct sb)

(auto split: memref.splits if-split-asm simp add: fun-upd-twist)
lemma flush-update-other: \(\forall m. a \notin \text{outstanding-refs (is-non-volatile-Write}_{sb} \Rightarrow \) 
flush sb (m(a:=v)) = (flush sb m)(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: 
\(\forall m. \forall j < \text{length ts}. \) 
(let \(-\cdot\cdot\cdot, sb_j\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\) = ts!j 
in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} \Rightarrow \) 
flush-all-until-volatile-write ts (m(a := v)) = 
(flush-all-until-volatile-write ts m)(a := v) 
proof (induct ts) 
  case Nil thus ?case by simp 
  next 
    case (Cons t ts) 
    note notin = \(\forall j < \text{length ts}. \) 
    (let \(-\cdot\cdot\cdot, sb_j\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\) = (t#ts)!j 
in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} \Rightarrow \) 
hence notin': \(\forall j < \text{length ts}. \) 
(let \(-\cdot\cdot\cdot, sb_j\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\) = ts!j 
in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} \Rightarrow \) 
by auto 

obtain p \_ \_ O_1 D_1 x_1 sb \_ \_ where t: t = (p_1, i_1, x_1, s_1, D_1, O_1, R_1) 
by (cases t) 

have no-write: 
  \(\text{outstanding-refs (is-volatile-Write}_{sb} \right\rangle \) 
  \(\text{takeWhile (Not o is-volatile-Write}_{sb} \right\rangle \) sb_j) = \{ \} 
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 is-non-volatile-Write}_{sb} \Rightarrow \) 
by (auto ) 

from flush-update-other' [OF a-notin no-write] 
have (flush (takeWhile (Not o is-volatile-Write}_{sb} \right\rangle \) sb_j) (m(a := v))) = 
(flush (takeWhile (Not o is-volatile-Write}_{sb} \right\rangle \) sb_j) m)(a := v). 
with Cons.hyps [OF notin', of (flush (takeWhile (Not o is-volatile-Write}_{sb} \right\rangle \) sb_j) 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: outstanding-refs is-volatile-Write @sb sb = {}
  shows \( \forall m. \{ i < \text{length } ts; ts[i] = (p, is, xs, sb, D, O, R) \}\)
  \( \land \) \( \forall j < \text{length } ts. i \neq j \rightarrow \)
  \( \text{let } (-, -, -, sb[j], -) = ts[j] \)
  \( \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write})_{sb} j)) \)
  \( \Rightarrow \) \( \text{flush-all-until-volatile-write } (ts[i := (p', is', xs, sb @ [Write sb False a sop v A L R W], D', O, R'])) m = \)
  \( \left( \text{flush-all-until-volatile-write } ts m \right) (a := v) \)

proof (induct ts)
  case Nil thus ?case
    by simp
  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 = \( \forall j < \text{length } (t#ts). i \neq j \rightarrow \)
    \( \text{let } (-, -, -, sb[j], -) = (t#ts)[j] \)
    \( \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write})_{sb} j)) \)
    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 have take-all: \( \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb = sb \)
          by (auto simp add: outstanding-refs-conv)
        from no-volatile-Write sb have take-all': \( \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) (sb @ [Write sb False a sop v A L R W]) = \)
          \( (sb @ [Write sb False a sop v A L R W]) \)
          by (auto simp add: outstanding-refs-conv)
        from notin have \( \forall j < \text{length } ts. \)
          \( \text{let } (-, -, -, sb[j], -) = ts[j] \)
          \( \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb[j]) \)
          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 \( p_1, is_1, O_1, R_1, D_1, xs_1, sb_1 \) where t: \( t = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1) \)
        by (cases t)
      from i-bound ith notin
      have flush-all-until-volatile-write
(ts[n := (p', is', xs, sb @ [Write_{sb} False a sop v A L R W], D', O, R)])
(flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) m) =
(flush-all-until-volatile-write ts
   (flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) m))
(a := v)
apply −
apply (rule Cons.hyps)
apply (auto simp add: Suc simp del: o-apply)
done

then
show ?thesis
  by (simp add: t Suc del: fun-upd-apply)
qed
qed

lemma flush-all-until-volatile-write-append-unflushed:
assumes volatile-Write_{sb}: ¬ outstanding-refs is-volatile-Write_{sb} sb = {}
shows □m i. [i < length ts; ts!i = (p, is, xs, sb, D, O, R)]
  ⇒ flush-all-until-volatile-write (ts[i := (p', is', xs, sb @ sbx, 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 < length (l#ts)⟩
  note ith = ⟨(l#ts)!i = (p, is, xs, sb, D, O, R)⟩:
  show ?case
    proof (cases i)
      case 0
      from ith 0 have l: l = (p, is, xs, sb, D, O, R)
        by simp
      from volatile-Write_{sb}
      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-append1 [of (Not ◦ is-volatile-Write_{sb}) r ] volatile-r
      have (flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb @ sbx) m) =
        (flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb ) m)
        by auto
      then
      show ?thesis
        by (simp add: 0 l)
    next
      case (Suc n)
      obtain p! is! O! R! D! xs! sb! where l: l = (p!, is!, xs!, sb!, D!, O!, R!)
        by (cases l)
      from Cons.hyps [of n] i-bound ith
show thesis
by (simp add: l Suc)
qed
qed

lemma flush-all-until-volatile-nth-update-unused:
shows \( m \ i. \ [i < \text{length ts}; \ ts!i = (p,\text{is},\text{sb},D,\mathcal{O},\mathcal{R})] \implies \text{flush-all-until-volatile-write} (ts[i := (p',\text{is}',\text{sb},D',\mathcal{O}',\mathcal{R}'])] m = (\text{flush-all-until-volatile-write} ts m) \)
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 = \( \langle (l#ts)!i = (p,\text{is},\theta,\text{sb},D,\mathcal{O},\mathcal{R})] \rangle \)

show \?case
proof (cases i)
case 0
from ith 0 have l: l = (p,\text{is},\theta,\text{sb},D,\mathcal{O},\mathcal{R})
by simp
show \?thesis
by (simp add: l)
next
case (Suc n)

obtain p_{\ell}, is_{\ell}, O_{\ell}, D_{\ell}, O_{\ell}, R_{\ell} where l: l = (p_{\ell},is_{\ell},\theta_{\ell},\text{sb}_{\ell},D_{\ell},O_{\ell},R_{\ell})
by (cases l)

from Cons.hyps [of n] i-bound ith
show \?thesis
by (simp add: l Suc)
qed
qed

lemma flush-all-until-volatile-write-append-volatile-write-commute:
assumes no-volatile-Write_{sb}: outstanding-refs is-volatile-Write_{sb} \{\}
shows \( m \ i. \ [i < \text{length ts}; \ ts!i = (p,\text{is},\text{sb},D,\mathcal{O},\mathcal{R})] \implies \text{flush-all-until-volatile-write} (ts[i := (p',\text{is}',\text{sb},D',\mathcal{O}',\mathcal{R}')] m = (\text{flush-all-until-volatile-write} ts m) \)
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 = \( \langle (l#ts)!i = (p,\text{is},\text{sb},D,\mathcal{O},\mathcal{R})] \rangle \)

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_{ab}
have s1: takeWhile (Not \circ is-volatile-Write_{ab}) sb = sb
  by (auto simp add: outstanding-refs-conv)

from no-volatile-Write_{ab}
have s2: (takeWhile (Not \circ is-volatile-Write_{ab}) (sb @ [Write_{ab} 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 \_ \_ \_ \_ \_ \_ \_ \_ where
  l = (p, is, \theta, sb, D, O, R)
  by (cases l)

from Cons.hyps [of n] i-bound ith
show \(?thesis
  by (simp add: l Suc)
qed

lemma reads-consistent-update:
\( \forall \text{pending-write } O \: m. \text{reads-consistent pending-write } O \: m \: sb \Longrightarrow \)
  a \notin \text{outstanding-refs} (Not \circ \text{is-volatile}) sb \Longrightarrow
  \text{reads-consistent pending-write } O \: (m(a := v)) \: sb

apply (induct sb)
apply simp
apply (clarsimp split: memref.splits if-split-asm
  simp add: fun-upd-twist)
subgoal for sb O m x11 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:
\( \forall p. \text{history-consistent } \theta \: p \: x \)
  \Longrightarrow \text{history-consistent } \theta \: (hd-prog \: p \: x) \: x

apply (induct xs)
apply simp
apply (auto split: memref.splits option.splits)
done
done

locale valid-program = program +
fixes valid-prog
assumes valid-prog-inv: \[ \theta \vdash p \rightarrow p' \: (p', is') \]: valid-prog p \Longrightarrow valid-prog p'
lemma (in valid-program) history-consistent-appendD:
\[ \forall \theta \ ys \ p. \ \forall \ sop \in \ write-sops \ xs. \ valid-sop \ sop \ \Rightarrow \ read-tmps \ xs \cap \ read-tmps \ ys = \{\} \ \Rightarrow \]
\[ \text{history-consistent} \ \theta \ p \ (xs@ys) \ \Rightarrow \]
\[ \text{(history-consistent} \ (\theta)^t \ (dom \ \theta - \ read-tmps \ ys)) \ p \ xs \ \land \]
\[ \text{history-consistent} \ \theta \ (last-prog \ p \ xs) \ ys \ \land \]
\[ \text{read-tmps} \ ys \cap \ \bigcup (\text{fst} \ i \ \text{write-sops} \ xs) = \{\} \]

proof (induct xs)
next
  case Nil thus ?case
    by auto
  case (Cons x xs)
  note valid-sops = \[ \forall \ sop \in \ write-sops \ (x \# \ xs). \ valid-sop \ sop \]
  note read-tmps-dist = \[ \text{read-tmps} \ (x \# \ xs) \cap \ read-tmps \ ys = \{\} \]
  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 where
      sop: sop = (D, f)
    by (cases sop)
  from consis obtain
    D-tmps: D \subseteq dom \ \theta \ \land
    f-v: f \ \theta = v \ \land
    D-read-tmps: D \cap \ read-tmps \ (xs \@ \ ys) = \{\} \ \land
    consis': \text{history-consistent} \ \theta \ p \ (xs \@ \ ys)
  by (simp add: Write sb sop)
  from valid-sops obtain
    valid-Df: valid-sop (D,f) \land
    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 read-tmps-dist have read-tmps-dist': \text{read-tmps} \ xs \cap \ read-tmps \ ys = \{\}
  by (simp add: Write sb)
  from D-read-tmps have D-ys: D \cap \ read-tmps \ ys = \{\}
  by (auto simp add: read-tmps-append)
  with D-tmps have D-subset: D \subseteq dom \ \theta - \ read-tmps \ ys
  by auto
  moreover
  from valid-sop [OF refl D-tmps]
  have f \ \theta = f \ (\ \theta \ |^t \ D),
  moreover
  let ?\theta' = \theta \ |^t \ (dom \ \theta - \ read-tmps \ ys)
  from D-subset
  have ?\theta' \ |^t \ D = \ \theta \ |^t \ D
  apply --
  apply (rule ext)
by (auto simp add: restrict-map-def)

moreover
from D-subset
have D-tmps': D ⊆ dom ?θ'
  by auto
ultimately
have f-v': f ?θ' = v
  using valid-sop [OF refl D-tmps'] f-v
  by simp
from D-read-tmps
have D ∩ read-tmps xs = {}
  by (auto simp add: read-tmps-append)
with Cons.hyps [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: ?θ t = Some v and
  consis': history-consistent ?θ p (xs @ ys)
  by (simp add: Read sb split: option splits)
from read-tmps-dist
obtain t-ys: t ∉ read-tmps ys and read-tmps-dist': read-tmps xs ∩ read-tmps ys = {}
  by (auto simp add: Read sb)
from valid-sops have valid-sops': ∀ sop∈write-sops xs. valid-sop sop
  by (auto simp add: Read sb)
from t-ys tmps-t
have (?θ |' (dom ?θ − read-tmps ys)) t = Some v
  by (auto simp add: restrict-map-def domIff)
with Cons.hyps [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: ?θ |' (dom ?θ − read-tmps (xs @ ys)) ⊢ p1 →p (p2, mis) and
  consis': history-consistent ?θ p2 (xs @ ys)
  by (auto simp add: Prog sb)
let ?θ' = ?θ |' (dom ?θ − read-tmps ys)
have eq: ?θ' |' (dom ?θ' − read-tmps xs) = ?θ |' (dom ?θ − read-tmps (xs @ ys))
  apply (rule ext)
  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: Prog sb)
from read-tmps-dist

210
obtain read-tmps-dist': read-tmps xs ∩ read-tmps ys = {}
  by (auto simp add: Prog sb)
from Cons.hyps [OF valid-sops' read-tmps-dist' consis'] p_1-p prog-step eq
show  thesis
  by (simp add: Prog sb)
next
case Ghost sb
  with Cons show  thesis
  by auto
qed

lemma (in valid-program) history-consistent-appendI:
  \( \forall \theta \ ys \ p. \ \forall \ sop \in \ write-sops \ xs. \ valid-sop \ sop \Longrightarrow \\
  \text{history-consistent} \ (\theta \ |' \ (\text{dom } \theta \ - \ \text{read-tmps } ys)) \ p \ xs \Longrightarrow \\
  \text{read-tmps } ys \ \cap \ \bigcup (\text{fst } \ ' \ \text{write-sops } xs) = \{\} \ \Longrightarrow \ \text{valid-prog } p \Longrightarrow \\
  \text{history-consistent} \ \theta \ p \ (xs@ys) \)
proof (induct xs)
  case Nil thus  ?case by simp
next
case (Cons x xs)
  note valid-sops = \langle \forall \ sop \in \ write-sops \ (x # xs). \ valid-sop \ sop \rangle
  note consis-xs = \langle \text{history-consistent} \ (\theta \ |' \ (\text{dom } \theta \ - \ \text{read-tmps } ys)) \ p \ (x # xs) \rangle
  note consis-ys = \langle \text{history-consistent} \ \theta \ (\text{last-prog } p \ (x # xs)) \ ys \rangle
  note dist = \langle \text{read-tmps } ys \ \cap \ \bigcup (\text{fst } \ ' \ \text{write-sops } (x # xs)) = \{\} \rangle
  note valid-p = \langle \text{valid-prog } p \rangle
show  thesis
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 \subseteq \text{dom } \theta \ - \ \text{read-tmps } ys \text{ and}
    f-v: f (\theta \ |' \ (\text{dom } \theta \ - \ \text{read-tmps } ys)) = v \ (\text{is } f \ ? \ \theta = v) \text{ and}
    D-read-tmps: D \cap \text{read-tmps } xs = \{\} \text{ and}
    consis': \text{history-consistent} \ (\theta \ |' \ (\text{dom } \theta \ - \ \text{read-tmps } ys)) \ p \ xs
    by (simp add: Write sb sop)
  from D-tmps D-read-tmps
  have D \cap \text{read-tmps } (xs @ ys) = \{\}
    by (auto simp add: read-tmps-append)
moreover
  from D-tmps have D-tmps': D \subseteq \text{dom } \theta \text{ and}
    by auto
moreover
  from valid-sops obtain
    valid-Df: valid-sop (D, f) \text{ and}
    valid-sops': \forall \ sop \in \text{write-sops } xs. \ valid-sop \ sop
    by (auto simp add: Write sb sop)

211
from valid-Df
interpret valid-sop (D,f).

from D-tmps
have tmps-eq: \[ \theta \upharpoonright ((\text{dom } \theta - \text{read-tmps } ys) \cap D) = \theta \upharpoonright D \]
  apply -
  apply (rule ext)
  apply (auto simp add: restrict-map-def)
done

from D-tmps
have \[ f ?\theta = f (?\theta \upharpoonright D) \]
  apply -
  apply (rule valid-sop [OF refl ])
  apply auto
done

with valid-sop [OF refl D-tmps'] f-v D-tmps

have \[ f \theta = v \]
  by (clarsimp simp add: tmps-eq)
moreover
from consis-ys have consis-ys': history-consistent \theta (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)

next
  case (Read sb volatile a t v)
from consis-xs obtain
  t-v: \[ (\theta \upharpoonright (\text{dom } \theta - \text{read-tmps } ys)) t = \text{Some } v \]
  and consis-xs': history-consistent \theta (last-prog p xs) p xs
  by (clarsimp simp add: Read sb split: option.splits)
from t-v have \theta t = \text{Some } v
  by (auto simp add: restrict-map-def split: if-split-asm)
moreover
from valid-sops obtain
  valid-sops': \forall \text{sop\in write-sops } xs. valid-sop sop
  by (auto simp add: Read sb)
from consis-ys have consis-ys': history-consistent \theta (last-prog p xs) ys
  by (auto simp add: Read sb)
from dist have dist': read-tmps ys \cap \bigcup (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 \upharpoonright (dom \theta - \text{read-tmps} \ ys)

from consis-xs obtain

\ p_1-p: p_1 = p \ \text{and}
\ prog-step: ?\theta \upharpoonright (dom ?\theta - \text{read-tmps} \ xs) \vdash p_1 \rightarrow_p (p_2, \text{mis}) \ \text{and}
\ consis': \text{history-consistent} \ ?\theta \ p_2 \ xs

by (auto simp add: Prog_{sb})

have eq: ?\theta \upharpoonright (dom ?\theta - \text{read-tmps} \ xs) = ?\theta \upharpoonright (dom ?\theta - \text{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 \upharpoonright (dom \ \theta - \text{read-tmps} \ (xs @ ys)) \vdash p_1 \rightarrow_p (p_2, \text{mis}) \ by simp

moreover

from valid-sops obtain

valid-sops': \ \forall \ \text{sop} \in \text{write-sops} \ xs. \ \text{valid-sop} \ \text{sop}

by (auto simp add: Prog_{sb})

from consis-ys have consis-ys': \text{history-consistent} \ ?\theta \ (\text{last-prog} \ p_2 \ xs) \ ys

by (auto simp add: Prog_{sb})

from dist have dist': \text{read-tmps} \ ys \cap \bigcup (\text{fst} \ ' \ \text{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:

\ \wedge \ \theta \ ys \ p. \ \forall \ \text{sop} \in \text{write-sops} \ xs. \ \text{valid-sop} \ \text{sop} \ \Rightarrow
\ \text{read-tmps} \ xs \cap \text{read-tmps} \ ys = \{\} \ \Rightarrow \ \text{valid-prog} \ p \ \Rightarrow
\ \text{history-consistent} \ ?\theta \ p \ (xs@ys) =
\ \text{(history-consistent} \ ?\theta \upharpoonright (dom \ ?\theta - \text{read-tmps} \ ys)) \ p \ xs \ \wedge
\ \text{history-consistent} \ ?\theta \ (\text{last-prog} \ p \ xs) \ ys \ \wedge
\ \text{read-tmps} \ ys \cap \bigcup (\text{fst} \ ' \ \text{write-sops} \ xs) = \{\})

apply rule

apply (rule history-consistent-appendD, assumption+)

apply (rule history-consistent-appendI)

apply auto
done

**lemma** instrs-takeWhile-dropWhile-conv:

\[ \text{instrs } \text{xs} = \text{instrs } \text{(takeWhile } P \text{ xs)} \oplus \text{instrs } \text{(dropWhile } P \text{ xs)} \]

**by** (induct xs) (auto split: memref.split)

**lemma** (in program) history-consistent-hd-prog-p:

\[ \forall P. \text{history-consistent } \theta \text{ p xs } \implies p = \text{hd-prog } p \text{ xs} \]

**by** (induct xs) (auto split: memref.split, option.split)

**lemma** instrs-append: \[ \forall \text{ys. instrs } \text{(xs@ys)} = \text{instrs } \text{xs} \oplus \text{instrs } \text{ys} \]

**by** (induct xs) (auto split: memref.split)

**lemma** prog-instrs-append: \[ \forall \text{ys. prog-instrs } \text{(xs@ys)} = \text{prog-instrs } \text{xs} \oplus \text{prog-instrs } \text{ys} \]

**by** (induct xs) (auto split: memref.split)

**lemma** prog-instrs-empty: \[ \forall r \in \text{set } \text{xs. } \neg \text{is-Prog}_{sb} r \implies \text{prog-instrs } \text{xs} = [] \]

**by** (induct xs) (auto split: memref.split)

**lemma** length-dropWhile [termination-simp]: \[ \text{length } \text{(dropWhile } P \text{ xs)} \leq \text{length } \text{xs} \]

**by** (induct xs) auto

**lemma** prog-instrs-filter-is-Prog_{sb}:

\[ \text{prog-instrs } \text{(filter } (\text{is-Prog}_{sb}) \text{ xs)} = \text{prog-instrs } \text{xs} \]

**by** (induct xs) (auto split: memref.split)

**lemma** Cons-to-snoc: \[ \exists y. x \# \text{xs} = (y @ [y]) \]

**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 sb1 sb2

**assume** sb: sb @ [Read_{sb} volatile a t v] = sb1 @ sb2

**from** causal-Read

**interpret** causal-program-history Read volatile a t \# is_{sb} sb .

**show** \[ \exists \text{is. instrs } sb_2 @ \text{is}_{sb} = \text{is } @ \text{prog-instrs } sb_2 \]

**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 = 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: sb2-snoc y instrs-append prog-instrs-append)
qed
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 ∃ 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 sb2-snoc: sb2=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: sb2-snoc y instrs-append prog-instrs-append)
qed
qed

lemma causal-program-history-Prog_{sb}:
assumes causal-Write: causal-program-history is_{sb} sb
shows causal-program-history (is_{sb}@mis) (sb @ [Prog_{sb} p_1 p_2 mis])
proof
  fix sb₁ sb₂
  assume sb: sb @ [Progₖₜ sb p₁ p₂ mis] = sb₁ @ sb₂
  from causal-Write
  interpret causal-program-history isₖₜ sb .
  show ?is. instrs sb₂ @ (isₖₜ @ 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 = Progₖₜ p₁ p₂ mis and sb: sb = sb₁@ys
    by simp

    from causal-program-history [OF sb] obtain is where
      instrs ys @ (isₖₜ @ mis) = is @ prog-instrs (ys@[Progₖₜ 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
qed

lemma causal-program-history-Ghost:
assumes causal-Ghostₖₜ: causal-program-history (Ghost A L R W # isₖₜ) sb
shows causal-program-history isₖₜ (sb@[Ghostₖₜ A L R W])
proof
  fix sb₁ sb₂
  assume sb: sb @ [Ghostₖₜ A L R W] = sb₁ @ sb₂
  from causal-Ghostₖₜ
  interpret causal-program-history Ghost A L R W # isₖₜ sb .
  show ?is. instrs sb₂ @ isₖₜ = 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 = Ghostₖₜ A L R W and sb: sb = sb₁@ys
    by simp

    from causal-program-history [OF sb] obtain is where
      instrs ys @ Ghost A L R W # isₖₜ = is @ prog-instrs ys
    by auto
    then show ?thesis
    by (simp add: sb₂-snoc y instrs-append prog-instrs-append)
lemma hd-prog-last-prog-end: \[ p = \text{hd-prog} \ p \ \text{sb} \land \text{last-prog} \ p \ \text{sb} = p_\text{sb} \] \implies p = \text{hd-prog} \ p_\text{sb} \ \text{sb}
by (induct \text{sb}) (auto split: memref.splits)

lemma hd-prog-idem: \text{hd-prog} (\text{hd-prog} \ p \ \text{xs}) \ \text{xs} = \text{hd-prog} \ p \ \text{xs}
by (induct \text{xs}) (auto split: memref.splits)

lemma last-prog-idem: \text{last-prog} (\text{last-prog} \ p \ \text{sb}) \ \text{sb} = \text{last-prog} \ p \ \text{sb}
by (induct \text{sb}) (auto split: memref.splits)

lemma last-prog-hd-prog-append:
last-prog (hd-prog \ p_\text{sb} \ (\text{sb}@\text{sb}')) \ \text{sb} = \text{last-prog} \ (\text{hd-prog} \ p_\text{sb} \ \text{sb}') \ \text{sb}
apply (induct \text{sb})
apply (auto split: memref.splits)
done

lemma last-prog-hd-prog: \text{last-prog} \ (\text{hd-prog} \ p \ \text{xs}) \ \text{xs} = \text{last-prog} \ p \ \text{xs}
by (induct \text{xs}) (auto split: memref.splits)

lemma last-prog-append-Read_{sb}:
\forall p. \text{last-prog} \ p \ (\text{sb}@[\text{Read}_{sb} \ \text{volatile} \ \text{a} \ \text{t} \ \text{v}]) = \text{last-prog} \ p \ \text{sb}
by (induct \text{sb}) (auto split: memref.splits)

lemma last-prog-append-Write_{sb}:
\forall p. \text{last-prog} \ p \ (\text{sb}@[\text{Write}_{sb} \ \text{volatile} \ \text{a} \ \text{sop} \ \text{v} \ \text{A} \ \text{L} \ \text{R} \ \text{W}]) = \text{last-prog} \ p \ \text{sb}
by (induct \text{sb}) (auto split: memref.splits)

lemma last-prog-append-Prog_{sb}:
\forall x. \text{last-prog} \ x \ (\text{sb}@[\text{Prog}_{sb} \ p \ \text{p}' \ \text{mis}]) = \text{p}'
by (induct \text{sb}) (auto split: memref.splits)

lemma hd-prog-append-Prog_{sb}: \text{hd-prog} \ x \ (\text{sb}@[\text{Prog}_{sb} \ p \ \text{p}' \ \text{mis}]) = \text{hd-prog} \ p \ \text{sb}
by (induct \text{sb}) (auto split: memref.splits)

lemma hd-prog-last-prog-append-Prog_{sb}:
\forall p'. \text{hd-prog} \ p' \ \text{xs} = p' \implies \text{last-prog} \ p' \ \text{xs} = p_1 \implies
\text{hd-prog} \ p' (\text{xs}@[\text{Prog}_{sb} \ p_1 \ p_2 \ \text{mis}]) = p'
apply (induct \text{xs})
apply (auto split: memref.splits)
done

lemma hd-prog-append-Ghost_{sb}:
hd-prog p (sb@[Ghost A R L W]) = hd-prog p sb
by (induct sb) (auto split: memref.splits)

**lemma** last-prog-append-Ghost: sb
\[ \forall p. \text{last-prog p (sb @ [Ghost A L R W]) = last-prog p sb} \]
by (induct sb) (auto split: memref.splits)

**lemma** dropWhile-all-False-conv:
\[ \forall x \in \text{set xs. \neg P x \implies dropWhile P xs = xs} \]
by (induct xs) auto

**lemma** dropWhile-append-all-False:
\[ \forall y \in \text{set ys. \neg P y \implies dropWhile P (xs@ys) = dropWhile P xs @ ys} \]
apply (induct xs)
apply (auto simp add: dropWhile-all-False-conv)
done

**lemma** reads-consistent-append-first:
\[ \forall m ys. \text{reads-consistent pending-write } O m (xs @ ys) \implies \text{reads-consistent pending-write } O m xs \]
by (clarsimp simp add: reads-consistent-append)

**lemma** reads-consistent-takeWhile:
**assumes** consis: \( \text{reads-consistent pending-write } O m sb \)
**shows** \( \text{reads-consistent pending-write } O m (\text{takeWhile P sb}) \)
**using** reads-consistent-append [where \( xs=(\text{takeWhile P sb}) \) and \( ys=(\text{dropWhile P sb}) \)]
consis
apply (simp add: reads-consistent-append)
done

**lemma** flush-flush-all-until-volatile-write-Write-volatile-commute:
\[ \forall i. \text{flush-flush-all-until-volatile-write-Write sb} \]
\[ (\forall i. \text{flush-flush-all-until-volatile-write-Write sb}) \]
\[ ((\text{flush-all-until-volatile-write ts m})a := v) = \]
\[ \text{flush-all-until-volatile-write (ts[i := (p,is,xs, sb, D', O', R')])} \]
\[ (m(a := v)) \]
**proof** (induct ts)
**case** Nil thus ?case
by simp

next

case (Cons l ts)

note i-bound = \( i < \) length (l#ts)\)

note ith = \( ((l#ts)\!\! i = (p, is, xs, Write_{sb} True a sop v A L R W#sb, D, O, \mathcal{R})) \)

note disj = \( (\forall i < \) length (l#ts). \( (\forall j < \) length (l#ts). i \neq j \rightarrow (let (-, -, -, sb_i, -, -) = (l#ts)!i; (-, -, -, sb_j, -, -) = (l#ts)!j in outstanding-refs is-Write_{sb} sb_i \cap outstanding-refs is-Write_{sb} (takeWhile (Not \circ is-volatile-Write_{sb}) sb_j) = \{})))\)

note a-notin = \( (\forall j < \) length (l#ts). i \neq j \rightarrow (let (-, -, -, sb_j, -, -) = (l#ts)!j in a \notin \bigcup (\lambda (-, -, -, sb_j, -, -). \text{outstanding-refs is-Write}_{sb} (takeWhile (Not \circ is-volatile-Write_{sb}) sb_j) \}) \)

show ?case

proof (cases i)

case 0
from ith 0 have l: l = (p, is, xs, Write_{sb} True a sop v A L R W#sb, D, O, \mathcal{R})
by simp

have a-notin-ts:
  \( a \notin \bigcup \lambda (-, -, -, sb_j, -, -). \text{outstanding-refs is-Write}_{sb} (takeWhile (Not \circ is-volatile-Write_{sb}) sb_j) \) \( \mathcal{F} \) \( \mathcal{G} ?U \) j-bound: j < length ts and
jth: ts\!\! j = (p_{j}, is_{j}, O_{j}, R_{j}, D_{j}, x_{j}, sb_{j}) where
a-in-j: a \in outstanding-refs is-Write_{sb} (takeWhile (Not \circ 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 = outstanding-refs is-Write_{sb} (takeWhile (Not \circ is-volatile-Write_{sb}) sb)

have U-SB-disj: \( ?U \cap \) SB = \{\}
proof
{
fix a'
assume a'-in-U: a' ∈ ?U
have a' ∉ ?SB
proof
assume a'-in-SB: a' ∈ ?SB
hence a'-in-SB': a' ∈ 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 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
a'-in-j: a' ∈ outstanding-refs is-Write sb (takeWhile (Not ◦ 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 1
show False
by auto
qed
}
moreover
{
fix a'
assume a'-in-SB: a' ∈ ?SB
hence a'-in-SB': a' ∈ 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' ∉ ?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, 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 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 ts 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 \_is \_l \_R \_l \_O \_l \_D \_xs \_l \_sb \_l
  where l: l = (\_p, \_is, \_xs, \_sb, \_D, \_l, \_O, \_R)
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) \_sb)(a := v)))
  (a := v)) =
   flush-all-until-volatile-write (ts[n := (\_p, \_is, \_xs, \_sb, \_D, \_O, \_R])]
   ((flush (takeWhile (Not \circ is-volatile-Write \_sb) \_sb) \_sb)(a := v))
apply –
apply (rule Cons.hyps)
apply (force simp add: Suc Let-def simp del: o-apply)+
done

moreover
let \( \mathsf{SB} = \text{outstanding-refs is-Write}_{\mathsf{s}} \) \((\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\mathsf{s}}) \mathsf{s})\)

have \( a \notin \mathsf{SB} \)
proof
  assume \( a \in \mathsf{SB} \)
  with a-notin [rule-format, of 0]
  show \( \text{False} \)
by (auto simp add: l Suc)
qed
then
have ((flush (takeWhile (Not \( \circ \) \text{is-volatile-Write}_{\mathsf{s}}) \mathsf{s}) m)(a := v)) =
  (flush (takeWhile (Not \( \circ \) \text{is-volatile-Write}_{\mathsf{s}}) \mathsf{s}) (m(a := v)))
apply –
apply (rule update-commute’ [where \( m=m \) \text{and} \( F=\{a\} \) \text{and} \( G=\mathsf{SB} \)])
apply (auto intro: flush-unchanged-addresses
  flushed-values-mem-independent simp del: o-apply)
done
ultimately
show \( \text{thesis} \)
by (simp add: l Suc del: fun-upd-apply o-apply)
qed
qed

lemma (in program)
\( \forall \mathsf{sb}’. \text{history-consistent } \theta \ (\text{hd-prog } p \ (\mathsf{sb} @ \mathsf{sb}’)) \ (\mathsf{sb} @ \mathsf{sb}’) \implies\)
  \( \text{last-prog } p \ (\mathsf{sb} @ \mathsf{sb}’) = p \implies\)
last-prog (hd-prog p (sb@sb')) sb = hd-prog p sb'
proof (induct sb)
case Nil thus \(?\text{case} \) by simp
next
case (Cons r \( \mathsf{sb} \))
  have consis: \( \text{history-consistent } \theta \ (\text{hd-prog } p \ ((r \ # \ \mathsf{sb} \) @ \mathsf{sb}’)) \ ((r \ # \ \mathsf{sb} \) @ \mathsf{sb}’) \)
    by fact
  have \( \text{last-prog } p \ ((r \ # \ \mathsf{sb} \) @ \mathsf{sb}’) = p \) by fact
  show \(?\text{case} \)
  proof (cases \( r \))
    case Write\( \mathsf{sb} \) with Cons show \(?\text{thesis} \) by auto
next
case Read\( \mathsf{sb} \) with Cons show \(?\text{thesis} \) by (auto split: option.splits)
next
case (Prog\( \mathsf{sb} \) \( p1 \) \( p2 \) is)
  from \( \text{last-prog } \text{have } \text{last-prog-p2: } \text{last-prog } p2 \ (\mathsf{sb} \ @ \mathsf{sb}’) = p \)
by (simp add: Prog sb)
from consis obtain consis': history-consistent \( \emptyset \ p_2 \ (sb_1 @ sb') \)
  by (simp add: Prog sb)

hence history-consistent \( \emptyset \ (hd-prog p_2 \ (sb_1 @ sb')) \ (sb_1 @ sb') \)
  by (rule history-consistent-hd-prog)
from Cons.hyps [OF this ] have last-prog p_2 sb_1 = hd-prog p sb'
oops

lemma last-prog-to-last-prog-same: \( \forall p'. \ last-prog p' \ sb = p \implies last-prog p \ sb = p \)
  by (induct sb) (auto split: memref.splits)

lemma last-prog-hd-prog-same: [last-prog p' \ sb = p; hd-prog p' \ sb = p'] \implies hd-prog p
  sb = p'
  by (induct sb) (auto split : memref.splits)

lemma last-prog-hd-prog-last-prog:
  last-prog p' \ (sb@sb') = p \implies hd-prog p' \ (sb@sb') = p' \implies
  last-prog (hd-prog p sb') \ sb = last-prog p' \ sb
apply (induct sb)
apply (simp add: last-prog-hd-prog-same)
apply (auto split : memref.splits)
done

lemma (in program) last-prog-hd-prog-append':
\( \forall sb' p. \ history-consistent \emptyset \ (hd-prog p \ (sb@sb')) \ (sb@sb') \implies
  last-prog (hd-prog p sb') \ sb = hd-prog p \ sb' \)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons r sb_1)
  have consis: history-consistent \( \emptyset \ (hd-prog p ((r # sb_1) @ sb')) \ ((r # sb_1) @ sb') \)
    by fact
  have last-prog: last-prog p ((r # sb_1) @ sb') = p by fact
  show ?thesis by case
proof (cases r)
case Write_{sb} with Cons show ?thesis by auto
next
case Read_{sb} with Cons show ?thesis by (auto split: option.splits)
next
case (Prog_{sb} p_1 p_2 is)
  from last-prog have last-prog-p2: last-prog p_2 \ (sb_1 @ sb') = p
    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 \( \emptyset \ p_2 \ (sb_1 @ sb') \)
    by (simp add: Prog_{sb})
  from history-consistent-hd-prog-p [OF consis']
have hd-prog-p2: hd-prog p2 (sb1 @ sb') = p2 by simp
from consis have history-consistent ≠ (hd-prog p (sb1 @ sb')) (sb1 @ sb')
by (rule history-consistent-hd-prog)
from Cons.hyps [OF this last-prog-p]
have last-prog (hd-prog p sb') sb1 = hd-prog p sb'.
moreover
from last-prog-hd-prog-last-prog [OF last-prog-p2 hd-prog-p2]
have last-prog (hd-prog p sb') sb1 = last-prog p2 sb1.
ultimately
have last-prog p2 sb1 = hd-prog p sb'
by simp
thus ?thesis
by (simp add: Prog sb)
next
case Ghostsb with Cons show ?thesis by (auto split: option.splits)
qed

lemma flush-all-until-volatile-write-Writeab-non-volatile-commute:
\[ \begin{array}{l}
\forall i. \forall m. \forall t. i < \text{length } t; \text{tsli}=(p,\text{is},\text{xs},\text{Writeab False a sop v A L R W sb',D,O,R}); \\
\quad \forall i. \forall t. (\forall j. i \neq j \rightarrow \\
\quad \quad (\text{let } (-,-,sb_i,-,-) = \text{tsli}; \\
\quad \quad (-,-,sb_j,-,-) = \text{tslj} \\
\quad \quad \text{in outstanding-refs is-Write}_{sb} (sb_j) \cap \\
\quad \quad \text{outstanding-refs is-Write}_{sb} \text{ (takeWhile (Not is-volatile-Write}_{sb} sb) sb_j) = \\
\quad \quad \{\}) ; \\
\quad \forall j. i \neq j \rightarrow \\
\quad \quad (\text{let } (-,-,sb_i,-,-) = \text{tsli in a } \notin \text{ outstanding-refs is-Write}_{sb} (\text{takeWhile (Not is-volatile-Write}_{sb} sb) sb_j) \}\} \\
\rightarrow \text{flush-all-until-volatile-write (ts[i := (p,\text{is},\text{xs},sb',D',O,R')]})(m(a := v)) = \\
\text{flush-all-until-volatile-write ts m}
\end{array} \]
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 = \[ \langle (l#ts)!i = (p,\text{is},\text{xs},\text{Writeab False a sop v A L R W sb',D,O,R}) \rangle \]
note disj = \[ \forall i. \forall t. i \neq j \rightarrow \\
\quad (\text{let } (-,-,sb_i,-,-) = (l#ts)i; \\
\quad (-,-,sb_j,-,-) = (l#ts)j \\
\quad \text{in outstanding-refs is-Write}_{ab} (sb_i) \cap \\
\quad \text{outstanding-refs is-Write}_{ab} \text{ (takeWhile (Not is-volatile-Write}_{ab} sb) sb_j) = \\
\quad \{\}) ; \\
\quad \forall j. i \neq j \rightarrow \\
\quad \quad (\text{let } (-,-,sb_i,-,-) = (l#ts)i; \\
\quad \quad (-,-,sb_j,-,-) = (l#ts)j \\
\quad \quad \text{in a } \notin \text{ outstanding-refs is-Write}_{ab} \text{ (takeWhile (Not is-volatile-Write}_{ab} sb) sb_j) \}\} \\
\rightarrow \text{flush-all-until-volatile-write (ts[i := (p,\text{is},\text{xs},\text{sb',D',O,R'})]})(m(a := v)) = \\
\text{flush-all-until-volatile-write ts m}
\]
from ith 0 have l: l = (p, is, xs, Write sb False a sop v A L R W # sb, D, O, R)
  by simp
thus ?thesis
  by (simp add: 0 del: fun-upd-apply)
next
case (Suc n)
obtain p l is l O 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 disj a-notin have
  flush-all-until-volatile-write (ts[n := (p, is, xs, sb, D', O, R')])
    ((flush (takeWhile (Not ° is-volatile-Write sb) sb l) m)(a := v)) =
  flush-all-until-volatile-write ts
    (flush (takeWhile (Not ° is-volatile-Write sb) sb l) m)
  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 ° is-volatile-Write sb) sb)
have a \notin ?SB
  proof
    assume a \in ?SB
    with a-notin [rule-format, of 0]
    show False
  qed
then
have ((flush (takeWhile (Not ° is-volatile-Write sb) sb l) m)(a := v)) =
  (flush (takeWhile (Not ° is-volatile-Write sb) sb l) (m(a := v)))
  apply —
  apply (rule update-commute'[where m=m and F={a} and G=?SB])
  apply (auto intro:
    flushed-addresses
    flushed-values-mem-independent simp del: o-apply)
done
ultimately
  show ?thesis
  by (simp add: Suc del: fun-upd-apply o-apply)
qed

lemma (in program) history-consistent-access-last-read':
\forall p. history-consistent \forall p (sb @ [Read sb volatile a t v]) \Longrightarrow
  \forall t = Some v
apply (induct sb)
apply (auto split: memref.splits option.splits)
done

lemma (in program) history-consistent-access-last-read:
  history-consistent θ p (rev (Read sb volatile a t v # sb)) ⇒ θ t = Some v
by (simp add: history-consistent-access-last-read')

lemma flush-all-until-volatile-write-Read sb-commute:
  ∀ i m. [ i < length ts; ts!i = (p, is, θ, Read sb volatile a t v # sb, D, O, R) ]
  ⇒ flush-all-until-volatile-write (ts[i := (p, is, θ, sb, 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 < length (l#ts)"
  note ith = "((l#ts)!i = (p, is, θ, Read sb volatile a t v # sb, D, O, R)"
  show ?case
  proof (cases i)
  case 0
  from ith 0 have l: l = (p, is, θ, Read sb volatile a t v # sb, D, O, R)
  by simp
  thus ?thesis
  by (simp add: Suc l)
  next
  case (Suc n)
  obtain p_i is_i O_i R_i D_i θ_i sb_i where l: l = (p_i, is_i, θ_i, sb_i, D_i, O_i, R_i)
  by (cases l)
  from i-bound ith
  have flush-all-until-volatile-write (ts[n := (p, is, θ, sb, D', O, R')])
    (flush (takeWhile (Not ◦ is-volatile-Write sb) sb_i)) m
  = flush-all-until-volatile-write ts
    (flush (takeWhile (Not ◦ is-volatile-Write sb) sb_i)) 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-Ghost sb-commute:
  ∀ i m. [ i < length ts; ts!i = (p, is, θ, Ghost sb A L W # sb, D, O, R) ]
  ⇒ flush-all-until-volatile-write (ts[i := (p, is, θ, sb, D', O, R')]) m = flush-all-until-volatile-write ts m
proof (induct ts)
lemma flush-all-until-volatile-write-Prog\sb-commute:
\[ \forall i \in \mathbb{N}. [i < \text{length} ts; ts!i = (p, is, \theta, \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D, O, R)] \rightarrow \text{flush-all-until-volatile-write} (ts[i := (p, is, \theta, D', O', R')]) \mbox{ m} = \text{flush-all-until-volatile-write} ts \mbox{ m} \]

proof (induct ts)
  case Nil thus ?case
    by simp
next
  case (Cons ts)
    note i-bound = \[ i < \text{length} (l#ts) \]
    note ith = \[ (l#ts)!i = (p, is, \theta, \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D, O, R) \]
    show ?case
      proof (cases i)
        case 0
        from ith 0 have \[ l = (p, is, \theta, \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D, O, R) \]
          by simp
        thus ?thesis
          by (simp add: 0 del: fun-upd-apply)
      next
        case (Suc n)
        obtain p l is l O l R l D l \theta l lb where \[ l = (p_l, is_l, \theta_l, lb, D_l, O_l, R_l) \]
          by (cases l)
        from i-bound ith have flush-all-until-volatile-write (ts[n := (p', is', \theta', \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D', O', R')])
          (flush (takeWhile (Not \circ is-volatile-Write_{\sb}) lb) m) =
          flush-all-until-volatile-write ts
          (flush (takeWhile (Not \circ is-volatile-Write_{\sb}) lb) 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\sb-commute:
\[ \forall i \in \mathbb{N}. [i < \text{length} ts; ts!i = (p, is, \theta, \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D, O, R)] \rightarrow \text{flush-all-until-volatile-write} (ts[i := (p, is, \theta, D', O', R')]) \mbox{ m} = \text{flush-all-until-volatile-write} ts \mbox{ m} \]

proof (induct ts)
  case Nil thus ?case
    by simp
next
  case (Cons ts)
    note i-bound = \[ i < \text{length} (l#ts) \]
    note ith = \[ (l#ts)!i = (p, is, \theta, \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D, O, R) \]
    show ?case
      proof (cases i)
        case 0
        from ith 0 have \[ l = (p, is, \theta, \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D, O, R) \]
          by simp
        thus ?thesis
          by simp
      next
        case (Suc n)
        obtain p l is l O l R l D l \theta l lb where \[ l = (p_l, is_l, \theta_l, lb, D_l, O_l, R_l) \]
          by (cases l)
        from i-bound ith have flush-all-until-volatile-write (ts[n := (p', is', \theta', \text{Prog}_{\sb} p_1 p_2 \text{ mis}_\sb, D', O', R')])
          (flush (takeWhile (Not \circ is-volatile-Write_{\sb}) lb) m) =
          flush-all-until-volatile-write ts
          (flush (takeWhile (Not \circ is-volatile-Write_{\sb}) lb) m)
          apply
          apply (rule Cons.hyps)
          apply (auto simp add: Suc l)
          done
      then show ?thesis
        by (simp add:Suc l)
      qed
qed
by (simp add: 0 del: fun-upd-apply)

next
  case (Suc n)
  obtain p l is l D l θ l sb l where l: l = (p l, is l, θ 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, θ, sb, D', O, R')])
  (flush (takeWhile (Not o is-volatile-Write sb) sb)) m =
  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

lemma flush-all-until-volatile-write-append-Prog sb-commute:
  \(\forall i m. [i < length ts; ts!i=(p, is, \theta, sb, D, O, R)] \implies flush-all-until-volatile-write (ts[i := (p_2, is\_mis, \theta, sb@[Prog sb p_1 p_2 mis], D', O, R')]) m = flush-all-until-volatile-write ts m\)
proof (induct ts)
  case Nil thus ?case
  by simp
next
  case (Cons l ts)
  note i-bound = \(i < length (l#ts)\)
  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-Prog sb del: fun-upd-apply)
next
  case (Suc n)
  obtain p l is l D l θ 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_2, is\_mis, \theta, sb@[Prog sb p_1 p_2 mis], D', O, R')])
  (flush (takeWhile (Not o is-volatile-Write sb) sb)) m =
flush-all-until-volatile-write ts
  (flush (takeWhile (Not o is-volatile-Write sb) sb) m)
apply –
apply (rule Cons.hyps)
apply (auto simp add: Suc l)
done

then show ?thesis
  by (simp add: Suc l)
qed
qed

lemma (in program) history-consistent-append-Progsb:
  assumes step: θ ⊢ p →p (p’ mis)
  shows  history-consistent θ (hd-prog p xs) xs \implies last-prog p xs = p \implies
          history-consistent θ (hd-prog p’ (xs@[Prog sb p p’ mis])) (xs@[Prog sb p p’ mis])
proof (induct xs)
case Nil with step show ?case by simp
next
case (Cons x xs)
  note consis = \langle history-consistent θ (hd-prog p (x # xs)) (x # xs) \rangle
  note last = \langle last-prog p (x#xs) = p \rangle
  show ?case
    proof (cases x)
      case Write sb with Cons show ?thesis by (auto simp add: read-tmps-append)
    next
      case Read sb with Cons show ?thesis by (auto split: option.splits)
    next
      case (Prog sb p1 p2 mis’)
        from consis obtain
          step: θ ‘(dom θ – read-tmps (xs@[Prog sb p p’ mis])) \vdash p1 \rightarrow p (p2, mis’) and
          consis’: history-consistent θ p2 xs
        by (auto simp add: Progsb read-tmps-append)
        from last have last-p2: last-prog p2 xs = p
          by (simp add: Progsb)
        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@[Prog sb p p’ mis]))
          (xs@[Prog sb p p’ mis]).
        from history-consistent-hd-prog [OF this]
        have history-consistent θ (hd-prog p2 (xs@[Prog sb p p’ mis]))
          (xs@[Prog sb p p’ mis]).
        moreover
        from history-consistent-hd-prog-p [OF consis’]

have \( p_2 = \text{hd-prog} p_2 \) \( \text{xs} \).
from \( \text{hd-prog-last-prog-append-} \text{Prog} \) \( \text{OF} \) this [symmetric] last-\( p_2 \)
have \( \text{hd-prog} p_2 \) \( (\text{xs} \ @ [\text{Prog} \ p \ p' \text{mis}]) = p_2 \)
  by simp
ultimately
have history-consistent \( \not\in \) \( p_2 \) \( (\text{xs} \ @ [\text{Prog} \ p \ p' \text{mis}]) \)
  by simp
thus \( \text{thesis} \)
  by (simp add: \( \text{Prog} \) step)
next
case Ghost \( \text{sb} \) with Cons show \( \text{thesis} \) by (auto)
qed
qed

primrec release :: 'a memref list \( \Rightarrow \) addr set \( \Rightarrow \) rels \( \Rightarrow \) rels
where
\( \text{release} [] \ S \ R = \ R \)
  | release (x#xs) \( \ S \ R = \)
  (case x of
    \( \text{Write} \) \( \text{sb} \) \( \text{volatile} \) - - \( A \ L \ R \ W \) \( \Rightarrow \)
      (if \( \text{volatile} \) then release \( \text{xs} \) \( (S \cup R - L) \) Map.empty
      else release \( \text{xs} \) \( S \ R \))
  | Ghost \( \text{sb} \) \( A \ L \ R \ W \) \( \Rightarrow \) release \( \text{xs} \) \( (S \cup R - L) \) (augment-rels \( \text{S} \ R \ R \))
  | - \( \Rightarrow \) release \( \text{xs} \) \( S \ R \))

lemma augment-rels-shared-exchange: \( \forall a \in R. \ (a \in S') = (a \in S) \Rightarrow \text{augment-rels} S \ R \ R = \text{augment-rels} S' \ R \ R \)
apply (rule ext)
apply (auto simp add: augment-rels-def split: option.splits)
done

lemma sharing-consistent-shared-exchange:
assumes shared-eq: \( \forall a \in \text{all-acquired sb}. \ S' a = S a \)
assumes consis: sharing-consistent \( S \ O \) \( \text{sb} \)
shows sharing-consistent \( S' O \) \( \text{sb} \)
using shared-eq consis
proof (induct \( \text{sb} \) arbitrary: \( S \ S' O \))
case Nil thus \( \text{case} \) by auto
next
case (Cons x sb)
show \( \text{case} \)
proof (cases x)
case (Write \( \text{sb} \) \( \text{volatile} \) \( \text{a sop v A L R W} \))
show \( \text{thesis} \)
proof (cases \( \text{volatile} \))
case True

from Cons.prems obtain
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': sharing-consistent \((S \oplus W R \ominus_A L) \ (O \cup A - R)\) sb and
shared-eq: \(\forall a \in A \cup \text{ all-acquired sb. } S' a = S a\)
by (clarsimp simp add: Write_sb True )
from shared-eq
have shared-eq': \(\forall a \in \text{ all-acquired sb. } (S' \oplus W R \ominus_A L) a = (S \oplus W R \ominus_A L) a\)
by (auto simp add: augment-shared-def restrict-shared-def)
from Cons.hyps [OF shared-eq' consis']
have sharing-consistent \((S' \oplus W R \ominus_A L) \ (O \cup A - R)\) 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
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': sharing-consistent \((S \oplus W R \ominus_A L) \ (O \cup A - R)\) sb and
shared-eq: \(\forall a \in A \cup \text{ all-acquired sb. } S' a = S a\)
by (clarsimp simp add: Ghost sb)
from shared-eq
have shared-eq': \(\forall a \in \text{ all-acquired sb. } (S' \oplus W R \ominus_A L) a = (S \oplus W R \ominus_A L) a\)
by (auto simp add: augment-shared-def restrict-shared-def)
from Cons.hyps [OF shared-eq' consis']
have sharing-consistent \((S' \oplus W R \ominus_A L) \ (O \cup A - R)\) 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: \(\forall a \in O \cup \text{ all-acquired sb. } S' a = S a\)
assumes consis: sharing-consistent \(S O\) sb
shows release sb (dom $S'$) $\mathcal{R}$ = release sb (dom $S$) $\mathcal{R}$

using shared-eq consis

proof (induct sb arbitrary: $S$ $S'$ $\mathcal{O}$ $\mathcal{R}$)

case Nil thus ?case by auto

next

case (Cons x sb)

to show ?case

proof (cases x)

case (Write$_{sb}$ volatile a sop v A L R W)

to show ?thesis

proof (cases volatile)

case True

from Cons.prems obtain

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

consis': sharing-consistent ($S \oplus_R \ominus_A L$) ($\mathcal{O} \cup A - R$) sb and

shared-eq: $\forall a \in \mathcal{O} \cup A \cup \text{all-acquired } sb. \ S' a = S a$

by (clarsimp simp add: Write$_{sb}$ True )

from shared-eq

have shared-eq': $\forall a \in \mathcal{O} \cup A - R \cup \text{all-acquired } sb. \ (S' \oplus_R \ominus_A L) a = (S \oplus_R \ominus_A L) a$

by (auto simp add: augment-shared-def restrict-shared-def)

from Cons.hyps [OF shared-eq' consis']

have release sb (dom ($S' \oplus_R \ominus_A L$)) Map.empty = release sb (dom ($S \oplus_R \ominus_A L$)) 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

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

consis': sharing-consistent ($S \oplus_R \ominus_A L$) ($\mathcal{O} \cup A - R$) sb and

shared-eq: $\forall a \in \mathcal{O} \cup A \cup \text{all-acquired } sb. \ S' a = S a$

by (clarsimp simp add: Ghost$_{sb}$ )

from shared-eq

have shared-eq': $\forall a \in \mathcal{O} \cup A - R \cup \text{all-acquired } sb. \ (S' \oplus_R \ominus_A L) a = (S \oplus_R \ominus_A L) a$

by (auto simp add: augment-shared-def 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 (augment-rels (dom $S'$) R $\mathcal{R}$) = (augment-rels (dom $S$) R $\mathcal{R}$).

with Cons.hyps [OF shared-eq' consis']
have release sb (dom ($S' \oplus W$ R $\ominus A$ L)) (augment-rels (dom $S'$) R $\mathcal{R}$) =
release sb (dom ($S \oplus W$ R $\ominus A$ L)) (augment-rels (dom $S$) R $\mathcal{R}$) by simp
then show ?thesis
  by (clarsimp simp add: Ghost sb domIff)
qed

lemma release-append:
$\forall S \mathcal{R}. \ \text{release} \ (sb@xs) \ (\text{dom } S) \mathcal{R} = \text{release} \ xs \ (\text{dom} \ (\text{share} \ sb \ S)) \ (\text{release} \ sb \ (\text{dom} \ (S)) \mathcal{R})$
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 Prog 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 $\mathcal{R}$]
  show ?thesis
  by (clarsimp simp add: Ghost sb)
qed

locale xvalid-program = valid-program +
fixes valid
assumes valid-implies-valid-prog:
[i < length ts;
\[ ts!i = (p, i, \theta, s, D, O, R); \text{valid } ts \] \implies \text{valid-prog } p \\

\textbf{assumes} valid-implies-valid-prog-hd:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, D, O, R); \text{valid } ts] \implies \text{valid-prog } (\text{hd-prog } p \ s) \]

\textbf{assumes} distinct-load-tmps-prog-step:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, D, O, R); \theta \vdash p \rightarrow p'(i, s')]; \text{valid } ts] \implies \\
\text{distinct-load-tmps } i' \land \\
(\text{load-tmps } i' \cap \text{load-tmps } s) = \{\} \land \\
(\text{load-tmps } i' \cap \text{read-tmps } s) = \{\} \]

\textbf{assumes} valid-data-dependency-prog-step:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, D, O, R); \theta \vdash p \rightarrow p'(i, s')]; \text{valid } ts] \implies \\
\text{data-dependency-consistent-instrs } (\text{dom } \theta \cup \text{load-tmps } s) = i' \land \\
\text{load-tmps } i' \cap \bigcup (\text{fst } \text{store-sops } s) = \{\} \land \\
\text{load-tmps } i' \cap \bigcup (\text{fst } \text{write-sops } s) = \{\} \]

\textbf{assumes} load-tmps-fresh-prog-step:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, D, O, R); \theta \vdash p \rightarrow p'(i, s')]; \text{valid } ts] \implies \\
\text{load-tmps } i' \cap \text{dom } \theta = \{\} \]

\textbf{assumes} valid-sops-prog-step:
\[ [\theta \vdash p \rightarrow p'(i, s')]; \text{valid-prog } p] \implies \forall \text{sop }\in\text{store-sops } i'. \text{valid-sop }\text{sop} \]

\textbf{assumes} prog-step-preserves-valid:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, D, O, R); \theta \vdash p \rightarrow p'(i, s')]; \text{valid } ts] \implies \\
\text{valid } (ts[i := (p', i, s, \theta, s, D, O, R)]) \]

\textbf{assumes} flush-step-preserves-valid:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, D, O, R); (m, s, b, O, R, S) \rightarrow f (m', s, b', O', R', S')]; \text{valid } ts] \implies \\
\text{valid } (ts[i := (p, i, \theta, s, b, D, O, R)]) \]

\textbf{assumes} sbh-step-preserves-valid:
\[ [i < \text{length } ts; ts!i = (p, i, \theta, s, b, D, O, R); (i, s, b, m, D, O, R, S) \rightarrow s bh (i, s, b', m, D', O', R', S')]; \text{valid } ts] \implies \\
\text{valid } (ts[i := (p, i, \theta, s, b, D', O', R)]) \]
lemma refl': x = y \Rightarrow r^{**} x y
  by auto

lemma no-volatile-Readsb-volatile-reads-consistent:
  \forall m. outstanding-refs is-volatile-Read_{sb} \Rightarrow volatile-reads-consistent m sb
apply (induct sb)
apply simp
subgoal for a sb m
apply (case-tac a)
apply (auto split: if-split-asm)
done
done

theorem (in program) flush-store-buffer-append:
shows \forall ts p m O R D S is O'.
[i < length ts;
  instrs (sb@sb') @ is_{sb} = is @ prog-instrs (sb@sb');
  causal-program-history is_{sb} (sb@sb');
  ts\!i = (p, is, \theta | (dom \theta - read-tmps (sb@sb')), x, D, O, R);
  p=hd-prog p_{sb} (sb@sb');
  (last-prog p_{sb} (sb@sb')) = p_{sb};
  reads-consistent True O' m sb;
  history-consistent \theta p (sb@sb');
  \forall sop \in write-sops sb. valid-sop sop;
  distinct-read-tmps sb;
  volatile-reads-consistent m sb
]\]

\Rightarrow \exists is'. instrs sb' @ is_{sb} = is' @ prog-instrs sb' \land
  (ts, m, S) \Rightarrow d^*
\Rightarrow (ts \!i = (last-prog (hd-prog p_{sb} sb') sb, is', \theta | (dom \theta - read-tmps sb'), x, (D \lor \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\}),
  acquired True sb O, release sb (dom S) R], flush sb m, share sb S)

proof (induct sb)
case Nil
thus ?case by (auto simp add: list-update-id' split: if-split-asm)
next
case (Cons r sb)
interpret direct-computation:
  computation direct-memop-step empty-storebuffer-step program-step \lambda p p' is sb. sb.
have ts-i:
  ts\!i = (p, is, \theta | (dom \theta - read-tmps ((r#sb)@sb')), x, D, O, R)
  by fact
have is: instrs ((r \# sb) @ sb') @ is_{sb} = is @ prog-instrs ((r \# sb) @ sb') by fact

have i-bound: i < length ts by fact
have causal: causal-program-history is\sb ((r \# sb) @ sb') by fact

hence causal': causal-program-history is\sb (sb @ sb')

by (auto simp add: causal-program-history-def)

note reads-consis = (reads-consistent True O' m (r#sb))

note p = (p = hd-prog psb ((r#sb)@sb'))

note psb = (last-prog psb ((r # sb) @ sb') = psb)

note hist-consis = (history-consistent \theta p ((r#sb)@sb'))

note valid-sops = (\forall sop \in write-sops (r#sb). valid-sop sop)

note dist = (distinct-read-tmps ((r#sb)@sb'))

note vol-read-consis = (volatile-reads-consistent m (r#sb))

show ?case

proof (cases r)
  case (Prog sb p1 p2 pis)

  from vol-read-consis
  have vol-read-consis': volatile-reads-consistent m sb
    by (auto simp add: Prog sb)

  from hist-consis obtain
    prog-step: \theta | (dom \theta \setminus read-tmps (sb @ sb')) |- p1 \rightarrow p (p2, pis) and
    hist-consis': history-consistent \theta p2 (sb @ sb')

    by (auto simp add: Prog sb)

  from p obtain p: p = p1

    by (simp add: Prog sb)

  from history-consistent-hd-prog [OF hist-consis']
  have hist-consis''': history-consistent \theta (hd-prog p2 (sb @ sb')) (sb @ sb')

  from is
  have is: instrs (sb @ sb') @ is\sb = (is @ pis) @ prog-instrs (sb @ sb')

    by (simp add: Prog sb)

  from ts-i is have
    ts-i: ts!i = (p, is, \theta | (dom \theta \setminus read-tmps (sb @ sb')), x, D, O,R)

    by (simp add: Prog sb)

  let ?ts' = ts[i:= (p2, is@pis, \theta | (dom \theta \setminus 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, S) \Rightarrow d (?ts', m, S) by simp

  also
  from i-bound have i-bound': i < length ?ts'

    by auto

  from i-bound
  have ts''-i: ?ts''!i = (p2, is@pis, (\theta | (dom \theta \setminus read-tmps (sb @ sb'))), x, D, O,R)

    by auto

236
from history-consistent-hd-prog-p [OF hist-consis.pk]

have p2-hd-prog: p2 = hd-prog p2 (sb @ sb').

from reads-consis have reads-consis': reads-consistent True O' m sb
  by (simp add: Prog sb)

from valid-sops have valid-sops': \forall sop \in write-sops sb. 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-p2: last-prog p2 (sb @ sb') = p sb
  by (simp add: Prog sb)

from hd-prog-last-prog-end [OF p2-hd-prog this]
have p2-hd-prog': p2 = hd-prog p sb (sb @ sb').

from last-prog-p2 [symmetric] have last-prog': last-prog p sb (sb @ sb') = p sb
  by (simp add: last-prog-idem)

from Cons.hyps [OF i-bound' ts':-i p2-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[i := (last-prog (hd-prog p sb sb') sb, is'),
     \emptyset |' (dom \emptyset - read-tmps sb'), x, D \lor outstanding-refs is-volatile-Write sb sb' \neq \{\},
     acquired True sb \emptyset, release sb (dom S) R],
     flush sb m, share sb S)
  by (auto)

from p sb have last-prog-eq: last-prog (hd-prog p sb sb') sb = last-prog p2 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 \ 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 \subseteq dom \emptyset and
\[ f \cdot \theta = v \quad \text{and} \]
\[ \text{dep: } D \cap \text{read-tmps (sb@sb')} = \{\} \quad \text{and} \]
\[ \text{hist-consist}: \text{history-consistent } \theta p (sb@sb') \]
\[ \text{by (simp add: Write}_{sb} \text{ sop split: option.splits)} \]

**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': \( \forall \text{sop } \in \text{write-sops sb. valid-sop sop} \)
\[ \text{by (simp add: Write}_{sb}) \]
interpret valid-sop sop **by** fact
**from** valid-sop [OF sop D-tmps]
have \( f \cdot \theta = f (\theta \mid' D) \).
moreover
**from** dep D-tmps have D-subset: \( D \subseteq (\text{dom } \theta - \text{read-tmps (sb@sb')}) \)
\[ \text{by auto} \]
moreover **from** D-subset have \( (\theta \mid' (\text{dom } \theta - \text{read-tmps (sb@sb')})) \mid' D = \theta \mid' D \)
apply –
apply (rule ext)
apply (auto simp add: restrict-map-def)
done
moreover **from** D-subset D-tmps have \( D \subseteq \text{dom } (\theta \mid' (\text{dom } \theta - \text{read-tmps (sb@sb')})) \)
\[ \text{by simp} \]
moreover
**note** valid-sop [OF sop this]
ultimately have \( f \cdot \theta = f (\theta \mid' (\text{dom } \theta - \text{read-tmps (sb@sb')})) = v \)
\[ \text{by (simp add: f-v)} \]
interpret causal': causal-program-history is\_sb sb@sb' **by** fact

**from** is
**have** Write volatile a sop A L R W\# instrs \((sb \@ sb') \@ is\_sb = is \@ prog-instrs (sb @ sb')\)
\[ \text{by (simp add: Write}_{sb}) \]
with causal',causal-program-history [of [], simplified, OF refl]
**obtain** is': where is: \( is=\text{Write \ volatile a sop A L R W\#is'} \quad \text{and} \)
is': instrs \((sb \@ sb') \@ is\_sb = is' \@ prog-instrs (sb @ sb')\)
\[ \text{by auto} \]

**from** ts-i is
**have** ts-i: tsli = (p,Write volatile a sop A L R W\#is',
\( \theta \mid' (\text{dom } \theta - \text{read-tmps (sb@sb')}),x,D,O,R) \)
\[ \text{by (simp add: Write}_{sb}) \]

**from** p **have** p': \( p = \text{hd-prog } p_{sb} (sb@sb') \)
\[ \text{by (auto simp add: Write}_{sb} \text{ hd-prog-idem)} \]

**from** p_{sb} **have** p_{sb}': last-prog p_{sb} (sb @ sb') = p_{sb}
\[ \text{by (simp add: Write}_{sb}) \]
show \(?\)thesis
proof (cases volatile)
  case False
    have memop-step:
      \(\text{(Write volatile a sop A L R W#is',}\theta|'(\text{dom } \theta - \text{read-tmps (sb@sb')}), x,m,\mathcal{D},\mathcal{O},\mathcal{R},\mathcal{S}) \rightarrow\)
      \((\text{is',}\theta|'(\text{dom } \theta - \text{read-tmps (sb@sb')}),x,m(a:=v),\mathcal{D},\mathcal{O},\mathcal{R},\mathcal{S})\)
    using D-subset
    apply (simp only: sop f-v [symmetric] False)
    apply (rule direct-memop-step. WriteNonVolatile)
    done

let \(?ts' = ts[i := (p, is',\theta |' (\text{dom } \theta - \text{read-tmps (sb @ sb')}),x, \mathcal{D}, \mathcal{O}, \mathcal{R})]\)
from direct-computation. Memop [OF i-bound ts-i memop-step]
have (ts, m, S) \&\& \(d(\text{ts'}, m(a := v), S)\).

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: \(?ts'! i = (p, is',\theta |' (\text{dom } \theta - \text{read-tmps (sb @ sb')}), x, \mathcal{D}, \mathcal{O}, \mathcal{R})\)
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) \Rightarrow_d^* 
(ts[i := (last-prog (\text{hd-prog p_{sb} sb') sb, is''), 
\theta |' (\text{dom } \theta - \text{read-tmps sb')}), x, 
\mathcal{D} \lor \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\}, acquired True sb \mathcal{O}, release sb
(dom S) \mathcal{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
  have memop-step:
    \(\text{(Write volatile a sop A L R W#is',}\theta|'(\text{dom } \theta - \text{read-tmps (sb@sb')}), x,m,\mathcal{D},\mathcal{O},\mathcal{R},\mathcal{S}) \rightarrow\)
    \((\text{is',}\theta|'(\text{dom } \theta - \text{read-tmps (sb@sb')}),x,m(a:=v),\mathcal{D},\mathcal{O} \cup A - \mathcal{R},\mathcal{Map}.\emptyset,\mathcal{S})\)
  using D-subset
apply (simp only: sop f-v' [symmetric] True)
apply (rule direct-memop-step.WriteVolatile)
done

let ?ts' = ts[i := (p, is', \theta |' (dom \theta - read-tmps (sb @ sb')), x, True, O U A - R, Map.empty)]
from direct-computation.Memop [OF i-bound ts-i memop-step]
have (ts, m, S) \Rightarrow_d (?ts', m(a := v), S +_W R \ominus_A L).

also
from reads-consis have reads-consis': reads-consistent True (O' U A - R)(m(a:=v))

by (auto simp add: Write sb True)

from i-bound have i-bound': i < length ?ts'
by auto

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 +_W R \ominus_A L)] i-bound
obtain is'' where
is'': instrs sb' @ is'_sb = is'' @ prog-instrs sb' and
steps: (?ts',m(a:=v),S +_W R \ominus_A L) \Rightarrow'_d
  (ts[i := (last-prog (hd-prog p'_sb sb')) sb], is''),
\theta |' (dom \theta - read-tmps sb'), x,
True, acquired True sb (O U A - R),release sb (dom (S +_W R \ominus_A L) Map.empty)],
flush sb (m(a := v)), share sb (S +_W R \ominus_A L)
by (auto simp del: fun-upd-apply)

note steps
finally
show ?thesis
by (simp add: Write sb True)
qed
next

case (Read sb volatile a t v)

from vol-read-consis reads-consis obtain v: v=m a and r-consis: reads-consistent True O' m sb and
vol-read-consis': volatile-reads-consistent m sb
by (cases volatile) (auto simp add: Read sb)

from valid-sops have valid-sops': \forall sop \in write-sops sb. valid-sop sop
by (simp add: Read sb)

from hist-consis obtain \theta: \theta t = Some v and
hist-consis': history-consistent \theta p (sb@sb')
by (simp add: Read sb split: option.splits)

from dist obtain t-notin: t \notin read-tmps (sb@sb') and
dist': distinct-read-tmps (sb@sb') by (simp add: Read sb)

from \emptyset t-notin have restrict-commute:
  \((\emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{read-tmps} (sb@sb')))(t \mapsto v) =
  \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{read-tmps} (sb@sb'))

  apply -
  apply (rule ext)
  apply (auto simp add: restrict-map_def dom_iff)
  done

from \emptyset t-notin

have restrict-commute':
  \(((\emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{insert} \ t \ (\mathrm{read-tmps} (sb@sb')))))(t \mapsto v) =
  \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{read-tmps} (sb@sb'))

  apply -
  apply (rule ext)
  apply (auto simp add: restrict-map_def dom_iff)
  done

interpret causal': causal-program-history is sb sb@sb' by fact

from is

have Read volatile a t # instrs (sb @ sb') @ is sb = is @ prog-instrs (sb @ sb')
  by (simp add: Read sb)

with causal'.causal-program-history [of [], simplified, OF refl]

obtain is' where is = Read volatile a t#is'

  is': instrs (sb @ sb') @ is sb = is' @ prog-instrs (sb @ sb')
  by auto

from ts-i is

have ts-i: tsli = (p, \mathrm{Read} \ \mathrm{volatile} \ a \ t#is',
  \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{insert} \ t \ (\mathrm{read-tmps} (sb@sb'))), x, D, O, R)

  by (simp add: Read sb)

from direct-memop-step.Read [of volatile a t is' \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{insert} \ t \ (\mathrm{read-tmps} (sb@sb')))] x m D O R S]

have memop-step:
  \((\mathrm{Read} \ \mathrm{volatile} \ a \ t \# \mathrm{is}',
  \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{insert} \ t \ (\mathrm{read-tmps} (sb @ sb'))), x, m, D, O, R, S) \rightarrow
  \mathrm{is}'

  \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{read-tmps} (sb @ sb'))), x, m, D, O, R, S)

  by (simp add: v [symmetric] restrict-commute restrict-commute')

let ?ts' = ts|ii := (p, is',

  \emptyset|\cdot (\mathrm{dom} \emptyset - \mathrm{read-tmps} (sb @ sb'))), x, D, O, R]

from direct-computation.Memop [OF i-bound ts-i memop-step]

have (ts, m, S) \Rightarrow_d (?ts', m, S).

241
also

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 auto

from p have p' : p = hd-prog p sb (sb@sb')
  by (auto simp add: Read sb hd-prog-idem)

from p sb have p sb' : last-prog p sb (sb @ sb') = p sb
  by (simp add: Read sb)

from Cons.hyps [OF i-bound' is' causal' ts'^i p' p sb' r-consis hist-consis'
valid-sops' dist' vol-read-consis']

obtain is'' where
  is'': instrs sb' @ is sb = is'' @ prog-instrs sb' and
  steps: (?ts'',m,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,share sb S)
  by (auto simp del: fun-upd-apply)

note steps

finally
show ?thesis
  using is''
  by (simp add: Read sb)

next
case (Ghost sb A L R W)

from vol-read-consis
have vol-read-consis' : volatile-reads-consistent m sb
  by (auto simp add: Ghost sb)

from reads-consis have r-consis: reads-consistent True (O' ∪ A − R) m sb
  by (auto simp add: Ghost sb)

from valid-sops have valid-sops' : ∀ sop ∈ write-sops sb. valid-sop sop
  by (simp add: Ghost sb)

from hist-consis obtain
  hist-consis' : history-consistent θ p (sb@sb')
  by (simp add: Ghost sb)
from dist obtain
dist': distinct-read-tmps (sb@sb') by (simp add: Ghost sb)

interpret causal': causal-program-history is sb sb@sb' by fact

from is
have Ghost A L R W# instrs (sb @ sb') @ is = is @ prog-instrs (sb @ sb')
  by (simp add: Ghost sb)

with causal'.causal-program-history [of [], simplified, OF refl]

obtain is' where is=Ghost A L R W#is' and
is': instrs (sb @ sb') @ is = is' @ prog-instrs (sb @ sb')
  by auto

from ts-i is
have ts-i: ts!i = (p, Ghost A L R W#is',
  \( \vartheta \mid (dom \vartheta - \text{read-tmps (sb@sb')}) \times D, O, R, S \))
  by (simp add: Ghost sb)

from direct-memop-step.Ghost [of A L R W is']
  \( \vartheta \mid (dom \vartheta - \text{read-tmps (sb@sb')}) \times m D O R S \)

have memop-step:
  (Ghost A L R W# is',\( \vartheta \mid (dom \vartheta - \text{read-tmps (sb@sb')}) \times x, m, D, O, R, S \))
  \( \rightarrow (is',\vartheta \mid (dom \vartheta - \text{read-tmps (sb@sb')}) \times x, m, D, O \cup A - R, \text{augment-rels (dom S) R R}) \)
  by auto

let ?ts' = ts[i := (p, is',
  \( \vartheta \mid (dom \vartheta - \text{read-tmps (sb@sb')}) \times D, O \cup A - R, \text{augment-rels (dom S) R R}) \)]

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',\( \vartheta \mid (dom \vartheta - \text{read-tmps (sb@sb')}) \times D, O \cup A - R, \text{augment-rels (dom S) R R}) \)
  by auto

from p have p': p = hd-prog p sb (sb@sb')
  by (auto simp add: Ghost sb hd-prog-idem)

from p sb have p sb': last-prog p sb (sb @ sb') = p sb
  by (simp add: Ghost sb)
from Cons.hyps \[ \text{OF i-bound' is' causal' ts' i p' p_
\text{sb}' r-consis hist-consis'} \]
valid-sops' dist' vol-read-consis', of \( S 
\oplus W \ R \ominus A \ L \]

obtain is'' where

is'" : instrs sb' @ is_{sb} = is'' @ prog-instrs sb' and

steps: (?ts',m,S \oplus W \ R \ominus A \ L) \Rightarrow^*_d

(\( \text{ts'}[i := (\text{last-prog (hd-prog p}_{sb}) \text{ sb'}) \text{ sb'}, \text{is'"},
\text{O} \land \text{read-tmps sb'},x), \text{D} \lor \text{outstanding-ref s-is-volatile-Write}_{sb} \neq \{\}, \text{acquired True sb (O} \cup A \land R),
\text{release sb (dom (S} \oplus W \ R \ominus A \ L)) (augment-rels (dom S) R \ R))],
flush sb m, share sb (S \oplus W \ R \ominus 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_{sb})

qed

qed

corollary (in program) flush-store-buffer:

assumes i-bound: i < length ts
assumes instrs: instrs sb @ is_{sb} = is @ prog-instrs sb
assumes cph: causal-program-history is_{sb} sb
assumes ts-i: ts!i = (p_{sb}, is_{sb}, \theta |' (dom \theta - read-tmps sb),x,D,O,R)
assumes p: p=hd-prog p_{sb} sb
assumes last-prog: (last-prog p_{sb} sb) = p_{sb}
assumes reads-consis: reads-consistent True O' m sb
assumes hist-consis: history-consistent \theta p sb
assumes valid-sops: \forall sop \in write-sops sb, valid-sop sop
assumes dist: distinct-read-tmps sb
assumes vol-read-consis: volatile-reads-consistent m sb

shows (ts,m,S) \Rightarrow^*_d

(\( \text{ts'}[i:=(p_{sb},is_{sb}, \theta,x), \text{D} \lor \text{outstanding-ref s-is-volatile-Write}_{sb} \neq \{\}, \text{acquired True sb O}, \text{release sb (dom S) R})),
flush sb m, share sb (S \oplus W \ R \ominus A \ L))

using flush-store-buffer-append [where sb'='[]], simplified, OF i-bound instrs cph ts-i
[simplified] p last-prog reads-consis hist-consis valid-sops dist vol-read-consis] last-prog

by simp

lemma last-prog-same-append: \( \forall xs \ p_{sb}. \ last-prog p_{sb} (sb@xs) = p_{sb} \Longrightarrow last-prog p_{sb} xs

= p_{sb}
apply (induct sb)
apply simp
subgoal for a sb xs p_{sb}
apply (case-tac a)
apply simp
apply simp

244
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:
\( \forall \text{pending-write } O. \forall m. \text{reads-consistent pending-write } O. \forall m. \text{sb } \implies \) outstanding-refs is-volatile-Read\(_{ab}\) ((dropWhile (Not is-volatile-Write\(_{ab}\)) sb) = {}}
apply (induct sb)
apply (auto split: memref.splits)
done

lemma reads-consistent-flush-other:
assumes no-volatile-Write\(_{ab}\)-sb: outstanding-refs is-volatile-Write\(_{ab}\) sb = {}}
shows \( \forall m. \text{pending-write } O. \) outstanding-refs (Not is-volatile-Read\(_{ab}\)) xs \( \cap \) outstanding-refs is-non-volatile-Write\(_{ab}\) sb = {}}: reads-consistent pending-write O m xs \( \implies \) reads-consistent pending-write O (flush sb m) xs
proof (induct xs)
case Nil thus \(?\)case by simp
next
case (Cons x xs)

note no-inter = \( \forall \)outstanding-refs (Not is-volatile-Read\(_{ab}\)) (x # xs) \( \cap \) outstanding-refs is-non-volatile-Write\(_{ab}\) sb = {}}

hence no-inter’: outstanding-refs (Not is-volatile-Read\(_{ab}\)) xs \( \cap \) outstanding-refs is-non-volatile-Write\(_{ab}\) sb = {}}
by (auto)

note consis = \( \forall \)reads-consistent pending-write O m (x # xs)
show ?case

proof (cases x)
case (Write\(_{ab}\) volatile a sop v A L R)
show ?thesis

proof (cases volatile)
case False

from consis obtain consis’: reads-consistent pending-write O (m(a := v)) xs
by (simp add: Write\(_{ab}\) 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\(_{ab}\) sb
by (auto simp add: Write\(_{ab}\) split: if-split-asm)

from flush-update-other’ [OF this no-volatile-Write\(_{ab}\)-sb]

245
have \((\text{flush } sb \ (m(a := v))) = (\text{flush } sb \ m)(a := v)\).

ultimately

show \textit{thesis}

by (simp add: \textit{Write}_{sb} False)

next

case True

from consis obtain consis'\': reads-consistent True \((O \cup A - R) \ (m(a := v)) \) \(xs\) \textbf{and} no-read: (outstanding-refs is-volatile-Read_{sb} \ xs = \{\} )

by (simp add: \textit{Write}_{sb} True)

from Cons.hyps [OF no-inter' consis']

have reads-consistent True \((O \cup A - R) \ (\text{flush } sb \ (m(a := v))) \) \(xs\).

moreover

from no-inter have a \notin outstanding-refs is-non-volatile-Write_{sb} sb

by (auto simp add: \textit{Write}_{sb} split: if-split-asm)

from flush-update-other' [OF this no-volatile-Write_{sb}-sb]

have \((\text{flush } sb \ (m(a := v))) = (\text{flush } sb \ m)(a := v)\).

ultimately

show \textit{thesis}

using no-read

by (simp add: \textit{Write}_{sb} True)

qed

next

case (Read_{sb} volatile a t v)

from consis obtain val: \((- \text{volatile } \rightarrow (\text{pending-write } \vee a \in O) \rightarrow v = m a)\) \textbf{and} consis'\': reads-consistent pending-write \(O \ m \ xs\)

by (simp add: \textit{Read}_{sb})

from Cons.hyps [OF no-inter' consis']

have hyp: reads-consistent pending-write \(O \ (\text{flush } sb \ m) \ xs\)

by simp

show \textit{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: \textit{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 \textit{thesis}

by (simp add: \textit{Read}_{sb})

next

with hyp val show \textit{thesis}

by (simp add: \textit{Read}_{sb})

qed

next

case Prog_{sb} with Cons show \textit{thesis} by auto
next
  case Ghost \_\_\_ with Cons show \?thesis by auto
qed
qed

lemma reads-consistent-flush-independent:
  assumes no-volatile-Write\_sb-sb: outstanding-refs is-Write\_sb sb \cap 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-sb
  have \( \forall a \in \text{outstanding-refs is-non-volatile-Read\_sb} \) 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 \( \bigwedge m \) pending-write \( O' \). [\text{reads-consistent pending-write} \( O' \) m xs; \( \forall i < \text{length ts} \).
  let \((p, is, \theta, sb, D, O, R) = ts!i \) in
  outstanding-refs (Not \circ\ is-volatile-Read\_sb) xs \cap
     outstanding-refs is-non-volatile-Write\_sb (takeWhile (Not \circ\ is-volatile-Write\_sb) sb) = \{
\] \Rightarrow \text{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 \theta\_t sb\_t
  where t: t=(p\_t, is\_t, \theta\_t, sb\_t, D\_t, O\_t, R\_t)
  by (cases t)
from Cons.prems t obtain
  no-inter: outstanding-refs (Not \circ\ is-volatile-Read\_sb) xs \cap
     outstanding-refs is-non-volatile-Write\_sb (takeWhile (Not \circ\ is-volatile-Write\_sb) sb\_t) = \{
  and
  no-inter': \( \forall i < \text{length ts} \).
  let \((p, is, \theta, sb, D, O, R) = ts!i \) in
  outstanding-refs (Not \circ\ is-volatile-Read\_sb) xs \cap
     outstanding-refs is-non-volatile-Write\_sb (takeWhile (Not \circ\ 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}) sb₁) = {}
   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}) sb₁) = {}
   by auto

from reads-consistent-flush-other [OF out1 this consis]

have reads-consistent pending-write \(O'\) (flush (takeWhile (Not ◦ is-volatile-Write_{sb}) sb₁) 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 \(\forall m \ O.\)
  [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)
   ) = {};
  reads-consistent False \(O\) m xs;
  read-only-reads \(O\) (takeWhile (Not ◦ is-volatile-Write_{sb}) xs) ⊆ RO]
  \(\Longrightarrow\) 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 = ⟨read-only-reads \(O\) (takeWhile (Not ◦ is-volatile-Write_{sb}) (x#xs)) ⊆ RO⟩
show \( \text{case} \)
proof (cases x)
  case (Write \(_{ab}\) volatile a sop v A L R)
  show \( \text{?thesis} \)
  proof (cases volatile)
    case False
    from consis obtain consis': reads-consistent False \( O \) (m(a := v)) xs
    by (simp add: Write \(_{ab}\) 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 \(_{ab}\) xs = \{\})
    by (simp add: Write \(_{ab}\) True)
  from no-inter obtain
  a-notin: a \( \notin \) outstanding-refs is-non-volatile-Write \(_{ab}\) sb and
  disj: (outstanding-refs (Not \circ is-volatile-Read \(_{ab}\)) xs) \( \cap \)
  outstanding-refs is-non-volatile-Write \(_{ab}\) sb = \{\}
  by (auto simp add: Write \(_{ab}\) True aargh misc-outstanding-refs-convs)
  from reads-consistent-flush-other [OF no-volatile-Write \(_{ab}\)-sb disj consis']
have reads-consistent True \((\mathcal{O} \cup A - R)\) (flush \(sb\) \((m(a := v))\)) \(xs\).

moreover

note \(a\)-notin

from flush-update-other [OF this no-volatile-Write\(_{sb}\)-\(sb\)]
have (flush \(sb\) \((m(a := v))) = (\text{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: \((\neg \text{volatile }\rightarrow a \in \mathcal{O} \rightarrow v = m a)\) and

consis': reads-consistent False \(\mathcal{O}\ m\ xs\)
by (simp add: Read\(_{sb}\))

from RO
have RO': read-only-reads \(\mathcal{O}\) (takeWhile (Not \circ \text{is-volatile-Write\(_{sb}\)}) \(xs\)) \subseteq \(RO\)
by (auto simp add: Read\(_{sb}\))

from no-inter
have no-inter': outstanding-refs is-non-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\(_{sb}\)}) \(xs\)) \(\cap\)
\(RO\) \(\cup\)
\((\mathcal{O} \cup \text{all-acquired} \) (takeWhile (Not \circ \text{is-volatile-Write\(_{sb}\)}) \(xs\)))
) = \{\}
by (fastforce simp add: Read\(_{sb}\) aargh)

show \(?\)thesis

proof (cases volatile)

case True

from Cons.hyps [OF no-inter' consis' RO']
show \(?\)thesis
by (simp add: Read\(_{sb}\) True)

next

case False

note non-volatile=this

from Cons.hyps [OF no-inter' consis' RO']
have hyp: reads-consistent False \(\mathcal{O}\) (flush \(sb\) \(m\)) \(xs\).

show \(?\)thesis
proof (cases a ∈ O)
case False
with hyp show ?thesis
  by (simp add: Read sb 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 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
next
case Progs 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-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 ∪ A − R ∪ all-acquired (takeWhile (Not ◦ is-volatile-Write sb) xs)) = {}
  by (fastforce simp add: Ghost sb aargh)
from Cons.hyps [OF no-inter' consis' RO']
show ?thesis
  by (clarsimp simp add: Ghost sb)

251
lemma reads-consistent-flush-all-until-volatile-write-aux':

assumes no-reads: outstanding-refs is-volatile-Read sb xs = {}
assumes read-only-reads-RO: read-only-reads O' (takeWhile (Not o is-volatile-Write sb) xs) ⊆ RO

shows \( \forall m. \ \& \ \& \ \text{reads-consistent False } O' m x; \ \forall i < \text{length ts.} \)

let (p,is,\theta ,sb,D,O) = ts\!\!/i

(outstanding-refs is-non-volatile-Write sb (takeWhile (Not o 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 o is-volatile-Write sb) xs) \cup
(RO \cup
(O' \cup \text{all-acquired (takeWhile (Not o is-volatile-Write sb) xs)}))

\( = \{ \} \)

\( \Rightarrow \) reads-consistent False O' (flush-all-until-volatile-write ts m) xs

proof (induct ts)

next

have consis: reads-consistent False O' m xs by fact

obtain p t is_t R_t D_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)

obtain

no-inter: outstanding-refs is-non-volatile-Write sb (takeWhile (Not o is-volatile-Write sb) sb\!\!/i) \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 o is-volatile-Write sb) xs) \cup
(RO \cup
(O' \cup \text{all-acquired (takeWhile (Not o is-volatile-Write sb) xs)}))

\( = \{ \} \) and

no-inter': \( \forall i < \text{length ts.} \)

let (p,is,\theta ,sb,D,O) = ts\!\!/i

(outstanding-refs is-non-volatile-Write sb (takeWhile (Not o 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} \ (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{xs})

\bigcup
\left(\text{outstanding-refs is-non-volatile-Read}_{sb} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{xs})\right)
\setminus \text{RO}
\bigcup
\left(\text{O'} \cup \text{all-acquired} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{xs})\right)
\bigcup
\left(\text{O'} \cup \text{all-acquired} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{xs})\right)

= \{\}

proof --
show \ ?thesis
apply (rule that)
using Cons.prems (2) [rule-format, of 0]
apply (clarsimp simp add: t)
apply clarsimp
using Cons.prems (2)
apply --
subgoal for i
apply (drule-tac x=Suc i in spec)
apply (clarsimp simp add: Let-def simp del: o-apply)
done
done
qed

have out1: outstanding-refs is-volatile-Write_{sb}
\ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}_t) = \{\}
by (auto simp add: outstanding-refs-conv dest: set-takeWhileD)

from reads-consistent-flush-other' [OF out1 no-inter consis read-only-reads-RO ]
have reads-consistent False \text{O'} \ (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}_t) \ \text{m}) \ \text{xs}.
from Cons.hyps [OF this no-inter' ]
show \ ?case
by (simp add: t)
qed

lemma in-outstanding-refs-cases [consumes 1, case-names Write_{sb} Read_{sb}]:
a \in \text{outstanding-refs P xs} \iff
(\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} \iff \text{P (Write}_{sb} \ \text{volatile a sop v A L R W)} \iff \text{C})
(\forall \text{volatile t v} . (\text{Read}_{sb} \ \text{volatile a t v}) \in \text{set xs} \iff \text{P (Read}_{sb} \ \text{volatile a t v)} \iff \text{C})
apply (clarsimp simp add: outstanding-refs-conv)
subgoal for x
apply (case-tac x)
apply fastforce+
done

lemma dropWhile-Cons: (dropWhile P xs) = x#ys \implies \neg P \, x
apply (induct xs)
apply (auto split: if-split-asm)
done

lemma reads-consistent-dropWhile:
reads-consistent pending-write O m (dropWhile (Not ◦ is-volatile-Write sb) sb) =
reads-consistent True O m (dropWhile (Not ◦ is-volatile-Write sb) sb)
apply (case-tac (dropWhile (Not ◦ is-volatile-Write sb) sb))
apply (simp only:)
apply simp
apply (frule dropWhile-Cons)
apply (auto split: memref.splits)
done

theorem
reads-consistent-flush-all-until-volatile-write:
\( \forall i \ m \ \text{pending-write. } [\text{valid-ownership-and-sharing } S \, ts; \ i < \text{length } ts; \ ts!i = (p, is, \theta, sb, D, O, R); \ \text{reads-consistent pending-write } O \ m \ sb ] \)
\implies \text{reads-consistent True (acquired True (takeWhile (Not ◦ is-volatile-Write sb) sb) O)
(flush-all-until-volatile-write ts m) (dropWhile (Not ◦ is-volatile-Write sb) sb)}$
proof (induct ts)
case Nil thus ?case by simp
next
case (Cons t ts)
  note i-bound = \( i < \text{length } (t \# ts) \)
  note ts-i = \( (t \# ts)!i = (p, is, \theta, sb, D, O, R) \)
  note consis = \( \text{reads-consistent pending-write } O \ m \ sb \)
  note valid = \( \text{valid-ownership-and-sharing } S \, t#ts \)
  then interpret valid-ownership-and-sharing S t#ts.
  from valid-ownership-and-sharing-tl [OF valid] have valid' : valid-ownership-and-sharing S ts.

obtain p_t is_t O_t R_t D_t \theta_t sb_t
  where t = (p_t, is_t, \theta_t, sb_t, D_t, O_t, R_t)
  by (cases t)
show ?case
proof (cases i)
case 0
  with ts-i t have sb-eq: sb=sb_t
    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\( s_b \) (takeWhile (Not \( o \) is-volatile-Write\( s_b \)) sb) \(\neq\) \{\})
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\( s_b \): outstanding-refs is-volatile-Read\( s_b \) (dropWhile (Not \( o \) is-volatile-Write\( s_b \)) sb) = \{\}.
hence outstanding-refs (Not \( o \) is-volatile-Read\( s_b \)) (dropWhile (Not \( o \) is-volatile-Write\( s_b \)) sb)

= outstanding-refs (\(\lambda s.\) True) (dropWhile (Not \( o \) is-volatile-Write\( s_b \)) sb)
by (auto simp add: outstanding-refs-conv)

have \(\forall i < \) length ts.
let \((p, \text{is}, \theta, D, O, R) = ts ! i \) in outstanding-refs (Not \( o \) is-volatile-Read\( s_b \)) (dropWhile (Not \( o \) is-volatile-Write\( s_b \)) sb) \(\cap\) outstanding-refs is-non-volatile-Write\( s_b \) (takeWhile (Not \( o \) is-volatile-Write\( s_b \)) sb)
\(\neq\) \{\}
proof
{ 
fix \(j\) \(p \_j\) \text{is}_j \(O \_j\) \(R \_j\) \(D \_j\) \(\theta \_j\) \(s_b \_j\) \(x\)
assume j-bound: \(j < \) length ts
assume ts-\(j\): ts[\(j\)] = \((p \_j, \text{is}_j, \theta_j, D \_j, O \_j, R \_j)\)
assume x-in-sb: \(x \in\) outstanding-refs (Not \( o \) is-volatile-Read\( s_b \)) (dropWhile (Not \( o \) is-volatile-Write\( s_b \)) sb)
assume x-in-j: \(x \in\) outstanding-refs is-non-volatile-Write\( s_b \) (takeWhile (Not \( o \) is-volatile-Write\( s_b \)) 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-until-volatile-write-aux [OF no-vol-Read\( s_b \) 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, θ, sb, D, O, R)
    by simp

  have reads-consistent False O (flush (takeWhile (Not ◦ is-volatile-Write sb) sb) m) sb
    proof
      have no-vW:
        outstanding-refs is-volatile-Write sb (takeWhile (Not ◦ is-volatile-Write sb) sb) = {}
        apply (clarsimp simp add: outstanding-refs-conv)
        apply (drule set-takeWhileD)
        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_{ab} xs
by (auto simp add: False Write_{ab})
  from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W'' ys zs
where xs=ys@Write_{ab} True a sop'' v'' A'' L'' R'' W''#zs
by auto
hence x#xs = (x#ys)@Write_{ab} 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_{ab} xs
by (auto simp add: volatile Write_{ab})
from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W''#zs
where xs=ys@Write_{ab} True a sop'' v'' A'' L'' R'' W''#zs
by auto
hence x#xs = (x#ys)@Write_{ab} True a sop'' v'' A'' L'' R'' W''#zs
by auto
thus ?thesis
by blast
next
case Read_{ab}
from a-in have a ∈ outstanding-refs is-volatile-Write_{ab} xs
by (auto simp add: Read_{ab})
from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W''#zs
where xs=ys@Write_{ab} True a sop'' v'' A'' L'' R'' W''#zs
by auto
hence x#xs = (x#ys)@Write_{ab} True a sop'' v'' A'' L'' R'' W''#zs
by auto
thus ?thesis
by blast
qed
next
case Prog_{ab}
from a-in have a ∈ outstanding-refs is-volatile-Write_{ab} xs
by (auto simp add: Prog_{ab})
from Cons.hyps [OF this] obtain sop'' v'' A'' L'' R'' W''#zs
where xs=ys@Write_{ab} True a sop'' v'' A'' L'' R'' W''#zs
by auto
hence \(x#xs = (x#ys)@Write_{sb}\) True a sop\(^{\prime\prime}\) v\(^{\prime\prime}\) A\(^{\prime\prime}\) L\(^{\prime\prime}\) R W\(^{\prime\prime}\)#zs
by auto
thus ?thesis
by blast
next
  case Ghost\(_{sb}\)
  from a-in have a \(\in\) outstanding-refs is-volatile-Write\(_{sb}\) xs
  by (auto simp add: Ghost\(_{sb}\))
  from Cons.hyps [OF this] obtain sop\(^{\prime\prime}\) v\(^{\prime\prime}\) A\(^{\prime\prime}\) L\(^{\prime\prime}\) R W\(^{\prime\prime}\)#zs
  where xs=ys@Write\(_{sb}\) True a sop\(^{\prime\prime}\) v\(^{\prime\prime}\) A\(^{\prime\prime}\) L\(^{\prime\prime}\) R W\(^{\prime\prime}\)#zs
  by auto
hence \(x#xs = (x#ys)@Write_{sb}\) True a sop\(^{\prime\prime}\) v\(^{\prime\prime}\) A\(^{\prime\prime}\) L\(^{\prime\prime}\) R W\(^{\prime\prime}\)#zs
by auto
thus ?thesis
by blast
qed
qed

lemma sharing-consistent-mono-shared:
\(\exists S\ S'\ O.\ \)
\(\text{dom}\ S \subseteq \text{dom}\ S' \implies\) sharing-consistent S O sb \(\implies\) sharing-consistent S' O 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 and B=S' ⊕ W 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 and B=S' ⊕ W and C=L in restrict-mono-aux)
apply blast
done
done
done

done

lemma sharing-consistent-mono-owns:
\(\exists O\ O'\ S.\ \)
\(O \subseteq O' \implies\) sharing-consistent S O sb \(\implies\) sharing-consistent S O' sb
apply (induct sb)
apply simp
subgoal for a sb O O' S
apply (case-tac a)
apply clarsimp
apply clarsimp

258
subgoal for volatile a D f v A L R W
apply (frule-tac A=\emptyset and B=\emptyset' and C=A in union-mono-aux)
apply (frule-tac A=\emptyset \cup A and B=\emptyset' \cup 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=\emptyset and B=\emptyset' and C=A in union-mono-aux)
apply (frule-tac A=\emptyset \cup A and B=\emptyset' \cup A and C=R in set-minus-mono-aux)
apply fastforce
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 \cup all-shared is else all-shared is)
    | Ghost\sb A L R W ⇒ R \cup all-shared is
    | - ⇒ all-shared is)

lemma sharing-consistent-all-shared:
∀ \S \O. sharing-consistent \S \O \sb ⇒ all-shared \sb \subseteq \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 fastforce
done

lemma sharing-consistent-share-all-shared:
∀ \S. \dom (share \sb \S) \subseteq \dom \S \cup 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 \( S \oplus W R \ominus A L \)]
  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 \( S \oplus W R \ominus A L \)]
  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:
  \( \bigwedge S \mathcal{O} \). sharing-consistent \( S \mathcal{O} \) sb \(\implies\) dom (share sb \( S \)) − dom \( S \subseteq \mathcal{O} \)
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 \( S=(S \oplus_W R \ominus_A L) \) and \( O=(O \cup A - R) \)] Cons.prems
  show ?thesis
  by (auto simp add: Write\textsubscript{sb} True)
qed
next
case Read\textsubscript{sb} with Cons show ?thesis by auto
next
case Prog\textsubscript{sb} with Cons show ?thesis by auto
next
case (Ghost\textsubscript{sb} A L R W)
  with Cons.hyps [where \( S=(S \oplus_W R \ominus_A L) \) and \( O=(O \cup A - R) \)] Cons.prems
  show ?thesis by auto
defun unshared-all-unshared:
  \( \land S \subseteq O. \) sharing-consistent \( S \subseteq O \) \( \Longrightarrow \) dom \( S \) – dom (share sb S) \( \subseteq \) all-unshared sb
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 \( S=(S \oplus_W R \ominus_A L) \) and \( O=(O \cup A - R) \)] Cons.prems
  show ?thesis
  by (auto simp add: Write\textsubscript{sb} True)
qed
next
case Read\textsubscript{sb} with Cons show ?thesis by auto
next
case Prog\textsubscript{sb} with Cons show ?thesis by auto
next
case (Ghost\textsubscript{sb} A L R W)
  with Cons.hyps [where \( S=(S \oplus_W R \ominus_A L) \) and \( O=(O \cup A - R) \)] Cons.prems
  show ?thesis by auto

261
lemma unshared-acquired-or-owned:
\( \forall S O. \text{sharing-consistent } S O sb \implies \text{all-unshared } sb \subseteq \text{all-acquired } sb \cup O \)
apply (induct sb)
apply simp
subgoal for a
apply (case-tac a)
apply auto+
done
done

lemma all-shared-acquired-or-owned:
\( \forall S O. \text{sharing-consistent } S O sb \implies \text{all-shared } sb \subseteq \text{all-acquired } sb \cup O \)
apply (induct sb)
apply simp
subgoal for a
apply (case-tac a)
apply auto+
done
done

lemma sharing-consistent-preservation:
\( \forall S S' O. \left[ \text{sharing-consistent } S O sb; \right.
\text{all-acquired } sb \cap \text{dom } S - \text{dom } S' = \{\};
\text{all-unshared } sb \cap \text{dom } S' - \text{dom } S = \{\}\left] \implies \text{sharing-consistent } S' O sb \right\)
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 \( (x # sb) \cap \text{dom } S - \text{dom } S' = \{\} \) by fact
  have new-cond: all-unshared \( (x # sb) \cap \text{dom } S' - \text{dom } S = \{\} \) 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
  next
    case True
    from consis obtain
    A: A \subseteq \text{dom } S \cup O \text{ and}
    L: L \subseteq A \text{ and}
    A-R: A \cap R = \{\} \text{ and}
R: R ⊆ O and
consis': sharing-consistent (S ⊕ W R ⊗ A L) (O ∪ A – R) sb
by (clarsimp simp add: Write₃b True)

from removed-cond obtain rem-cond: (A ∪ all-acquired sb) ∩ dom S ⊆ dom S' by
(clarsimp simp add: Write₃b 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: Write₃b 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: Write₃b True)
qed

next (Ghost₃b 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: Ghost₃b)

from removed-cond obtain rem-cond: (A ∪ all-acquired sb) ∩ dom S ⊆ dom S' by
(clarsimp simp add: Ghost₃b)
  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: Ghost₃b)
  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: 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 \emptyset 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
    \forall a \in all-acquired sb. S' a = S a
shows sharing-consis S' ts
proof
  fix i p is O R D \emptyset 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

264
from sharing-consistent-shared-exchange [OF dist-sb consis]
show ?thesis .
qed
qed

lemma all-acquired-takeWhile: all-acquired (takeWhile P sb) ⊆ all-acquired sb
proof –
from all-acquired-append [of takeWhile P sb dropWhile P sb]
show ?thesis
  by auto
qed

lemma all-acquired-dropWhile: all-acquired (dropWhile P sb) ⊆ all-acquired sb
proof –
from all-acquired-append [of takeWhile P sb dropWhile P sb]
show ?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 $S$) ⊆ $O \cup$ 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 ⊆ dom $S \cup O$.
  moreover
  from sharing-consistent-share-all-shared have dom (share sb $S$) ⊆ dom $S \cup$ all-shared sb.
  moreover
  from sharing-consistent-all-shared [OF consis] have all-shared sb ⊆ dom $S \cup O$.
  ultimately
  show ?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

lemma share-owns-shared:
  assumes consis: sharing-consistent $S \ O$ sb
  shows dom (share sb $S$) ⊆ $O \cup$ dom $S$
  using acquired-share-owns-shared [OF consis]
  by blast
lemma all-shared-append: all-shared (xs@ys) = all-shared xs ∪ all-shared ys
   by (induct xs) (auto split: memref.splits)

lemma acquired-union-notin-first:
   ⋀ pending-write A B. a ∈ acquired pending-write sb (A ∪ B) ⇒ a ∉ A ⇒ a ∈ 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 ∉ A and
   a-acq: a ∈ acquired pending-write (x # sb) (A ∪ 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
note volatile = this
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: Write sb volatile 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: Write sb volatile True)
next
case False
from a-acq have a-acq': a ∈ acquired True sb (A' − R)
   by (simp add: Write sb volatile False)
then
show ?thesis
by (simp add: Write sb volatile False)
qed
qed
next
case (Ghost sb 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: Ghost_{sb} 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: Ghost_{sb} True)
next
  case False
  from a-acq have a-acq': a ∈ acquired False sb (A ∪ B)
  by (simp add: Ghost_{sb} False)
  from Cons.hyps [OF this a-notin-A]
  show ?thesis
    by (simp add: Ghost_{sb} False)
  qed
  qed (insert Cons, auto)
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 @ Ghost_{sb} 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 ∈ all-acquired xs
        by (auto simp add: False Write_{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 @ Ghost₄₅ A L R W # zs ∧ a ∈ A) (is ?write ∨ ?ghost).

then
show ?thesis

proof
assume ?write
then
obtain sop'' a'' v'' A'' L'' R'' W'' ys zs
where xs=ys@Write₄₅ True a'' sop'' v'' A'' L'' R'' W''#zs and a-in: a ∈ A''
by auto
hence x#xs = (x#ys)@Write₄₅ True a'' sop'' v'' A'' L'' R'' W''#zs
by auto
thus ?thesis
using a-in
by blast
next
assume ?ghost
then obtain A'' L'' R'' W'' ys zs where
xs=ys@Ghost₄₅ A'' L'' R'' W''#zs and a-in: a ∈ A''
by auto
hence x#xs = (x#ys)@Ghost₄₅ A'' L'' R'' W''#zs
by auto
thus ?thesis
using a-in
by blast
qed

next
case True
note volatile = this
show ?thesis

proof (cases a ∈ A)

next

true

with a-in have a ∈ all-acquired xs
by (auto simp add: volatile Write₄₅)

from Cons.hyps [OF this]

have (∃sop a' v ys zs A L R W. xs = ys @ Write₄₅ True a' sop v A L R W # zs ∧ a ∈ A) ∨
     (∃A L R W ys zs. xs = ys @ Ghost₄₅ A L R W# zs ∧ a ∈ A) (is ?write ∨ ?ghost).

then
show ?thesis

proof
assume ?write
then
obtain sop'' a'' v'' A'' L'' R'' W'' ys zs
where xs=ys@Write₄₅ True a'' sop'' v'' A'' L'' R'' W''#zs and a-in: a ∈ A''
by auto
hence x#xs = (x#ys)@Write₄₅ 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

case True
then have x#xs=[]@(Write sb True a's sop v A L R W#xs)
by (simp add: Write sb volatile True)
thus ?thesis
using True
by blast
qed

next

case Read sb
from a-in have a ∈ all-acquired xs
by (auto simp add: Read 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''#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

269
using a-in
by blast
qed

next
case Progsb
from a-in have a ∈ all-acquired xs
  by (auto simp add: Progsb)
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
case (Ghost sb A L R W)
show ?thesis
proof (cases a ∈ A)
case False
  with a-in have a ∈ all-acquired xs
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
  case True
  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
  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 voluntary 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 (Write sb)
  case (Write sb voluntary a’ sop v A L R W)
  show ?thesis
  proof (cases a’=a)
    case False
    with a-in have a ∈ outstanding-refs is-Write sb xs
    by (auto simp add: Write sb)
from Cons.hyps [OF this] obtain sop'' volatile'' v'' A'' L'' R'' W'' ys zs

271
where \( xs = ys @ \text{Write}_{ab} \) volatile" a sop" v" A" L" R" W" #zs
by auto
  hence x#xs = (x#ys) @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
  thus ?thesis
by blast
next
case True
  then have x#xs = ||@ (Write_{ab} volatile a sop v A L R W #zs)
by (simp add: Write_{ab} True)
  thus ?thesis
by blast
qed
next
case Read_{ab}
from a-in have a \( \in \) outstanding-refs is-Write_{ab} xs
  by (auto simp add: Read_{ab})
from Cons.hyps [OF this] obtain sop" volatile" v" A" L" R" W" ys zs
  where xs = ys @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
hence x#xs = (x#ys) @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
  thus ?thesis
by blast
next
case Prog_{ab}
from a-in have a \( \in \) outstanding-refs is-Write_{ab} xs
  by (auto simp add: Prog_{ab})
from Cons.hyps [OF this] obtain sop" volatile" v" A" L" R" W" ys zs
  where xs = ys @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
hence x#xs = (x#ys) @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
  thus ?thesis
by blast
next
case Ghost_{ab}
from a-in have a \( \in \) outstanding-refs is-Write_{ab} xs
  by (auto simp add: Ghost_{ab})
from Cons.hyps [OF this] obtain sop" volatile" v" A" L" R" W" ys zs
  where xs = ys @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
hence x#xs = (x#ys) @ Write_{ab} volatile" a sop" v" A" L" R" W" #zs
by auto
  thus ?thesis
by blast
qed
qed

lemma outstanding-refs-is-Write_{ab}-union:

272
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] ] \implies r** x z
by auto

lemma r-rtranclp-rtranclp: [ [r x y; r** y z] ] \implies r** x z
by auto

lemma unshared-is-non-volatile-Write\_sb: \( \bigwedge S. \\
[non-volatile-writes-unshared S sb; a \in dom S; a \notin all-unshared sb] \implies \\
a \notin 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 \( S=(S \ominus W R \ominus A L) \) ] 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 \( S=(S \ominus W R \ominus A L) \) ] Cons.prems show ?thesis by auto
qed
qed

lemma outstanding-non-volatile-Read\_sb-acquired-or-read-only-reads:
\( \bigwedge O S \) pending-write.
[non-volatile-owned-or-read-only pending-write S O sb;
\[a \in \text{outstanding-refs is-non-volatile-Read}_{sb}\]  
\[\implies a \in \text{acquired-reads True } sb \cup a \in \text{read-only-reads } \mathcal{O}\ sb\]

**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 (\(S \oplus_W R \ominus_A L\)) (\(O \cup A - R\))] 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** Prog\(_{sb}\) with Cons show ?thesis **by** auto 

**next**

**case** (Ghost\(_{sb}\) \(A\) \(L\) \(R\) \(W\)) with Cons.hyps [of pending-write (\(S \oplus_W R \ominus_A L\)) \(O \cup A - R\)] Cons.prems

**show** ?thesis

**by** auto 

qed 

**qed**

**lemma** acquired-reads-union: \(\bigland\text{pending-writes A B.}\)

\[a \in \text{acquired-reads pending-writes sb (A } \cup B\); a \not\in A\] \(\implies a \in \text{acquired-reads pending-writes sb B}\)

**proof** (induct \(sb\))

**case** Nil **thus** ?case by simp 

**next**

**case** (Cons \(x\) \(sb\))

**show** ?case

**proof** (cases \(x\))

**case** (Write\(_{sb}\) volatile \(a'\) sop v \(A'\) \(L'\) \(R'\) \(W'\))

**show** ?thesis

**proof** (cases volatile)
case True
note volatile=this
proof (cases pending-writes)
case True
from Cons.prems obtain
  a-in: a ∈ acquired-reads True sb (A ∪ B ∪ A’ − R’)
  a-notin: a /∈ A
  by (simp add: Write sb volatile True)
have (A ∪ B ∪ A’ − R’) ⊆ (A ∪ (B ∪ A’ − R’))
  by auto
from acquired-reads-mono [OF this] a-in
have a ∈ acquired-reads True sb (A ∪ (B ∪ A’ − R’))
  by auto

from Cons.hyps [OF this a-notin]
have a ∈ acquired-reads True sb (B ∪ A’ − R’).
then show ?thesis
  by (simp add: Write sb volatile True)
next
case False
with Cons show ?thesis
  by (auto simp add: Write sb volatile False)
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 split: if-split-asm)
next
case Prog sb with Cons show ?thesis
  by (auto)
next
case (Ghost sb A’ L’ R’ W’)
show ?thesis
proof –
  from Cons.prems obtain
  a-in: a ∈ acquired-reads pending-writes sb (A ∪ B ∪ A’ − R’)
  a-notin: a /∈ A
  by (simp add: Ghost sb)
  have (A ∪ B ∪ A’ − R’) ⊆ (A ∪ (B ∪ A’ − R’))
    by auto
  from acquired-reads-mono [OF this] a-in
  have a ∈ acquired-reads pending-writes sb (A ∪ (B ∪ A’ − R’))
    by auto
  from Cons.hyps [OF this a-notin]
have \( a \in \text{acquired-reads pending-writes sb} \) \((B \cup A' - R')\).

then show \(?\text{thesis}\)
  by (simp add: Ghost\(_{sb}\))
qed
qed
qed

lemma non-volatile-writes-unshared-no-outstanding-non-volatile-Write\(_{sb}\):
\( \forall S S'. \wedge S S' \).
\[ \forall a \in \text{dom } S' - \text{dom } S. a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb} \]
\( \Rightarrow \) 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: non-volatile-writes-unshared \((S \oplus W R \ominus_A L)\) sb and
            no-refs-sb: \( \forall a \in \text{dom } S' - \text{dom } S. a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb} \)
            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 \( \notin \) outstanding-refs is-non-volatile-Write\(_{sb}\) 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: \( \forall a \in \text{dom } S' - \text{dom } S. a \notin \text{outstanding-refs is-non-volatile-Write}_{sb} \text{ sb} \)
        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 /∈ outstanding-refs is-non-volatile-Write sb
by auto
from Cons.hyps [OF unshared-sb this]
show ?thesis
by (simp add: Ghost sb)
qed
qed

theorem sharing-consis-share-all-until-volatile-write:
∀ S ts'. [ownership-distinct ts; sharing-consis S ts; length ts' = length ts;
∀ i < length ts.
(let (r,sb,O) = ts!i;
(r,sb',O') = ts'!i
in O' = acquired True (takeWhile (Not ◦ is-volatile-Write sb) O) ∧
sb' = dropWhile (Not ◦ is-volatile-Write sb))]
sharing-consis (share-all-until-volatile-write ts S) ts' ∧
dom (share-all-until-volatile-write ts S) − dom S ⊆
∪ ((λ(r,sb,O). O) ' set ts) ∧
dom S − dom (share-all-until-volatile-write ts S) ⊆
∪ ((λ(r,sb,O). all-acquired sb ∪ O) ' 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: ∀ i < length (t#ts).
    (let (r,sb,O) = (t#ts)!i;
    (r,sb',O') = ts'!i
    in O' = acquired True (takeWhile (Not ◦ is-volatile-Write sb) O) ∧
sb' = dropWhile (Not ◦ is-volatile-Write sb))
    by fact
  obtain p is O R D ∅ sb
    where t: t = (p,is,∅,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' ∅ sb'
    where t': t' = (p',is',∅,sb',D',O',R')
    by (cases t')
  from sim [rule-format, of 0] t t' ts'
  obtain O': O' = acquired True (takeWhile (Not ◦ is-volatile-Write sb) O) and
    sb': sb' = dropWhile (Not ◦ is-volatile-Write sb)
    by auto
  from sim ts'
have sim': ∀i < length ts.
    (let (\(-,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot\)) = ts!i;
     (\,-,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot\)) = ts'!i
    in \(O'\) = acquired True (takeWhile (Not \(\circ\) is-volatile-Write_{\cdot,\cdot} \(sb\)) \(O\) ∧
    \(sb'\) = dropWhile (Not \(\circ\) is-volatile-Write_{\cdot,\cdot} \(sb\)) 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 \(\mathcal{O}\ \mathcal{S}\ \mathcal{O}\) sb
    by fastforce
from sharing-consistent-takeWhile [OF this]
have consis': sharing-consistent \(\mathcal{S}\ \mathcal{O}\) (takeWhile (Not \(\circ\) is-volatile-Write_{\cdot,\cdot} \(sb\)) sb)
    by simp

let ?S' = (share (takeWhile (Not \(\circ\) is-volatile-Write_{\cdot,\cdot} \(sb\)) \(S\))
from freshly-shared-owned [OF consis']
have fresh-owned: dom ?S' − dom \(S\) ⊆ \(O\).
from unshared-all-unshared [OF consis'] unshared-acq-owned-or-owned [OF consis']
have unshared-acq-owned: dom \(S\) − dom ?S'
    ⊆ all-acquired (takeWhile (Not \(\circ\) is-volatile-Write_{\cdot,\cdot} \(sb\)) sb) ∪ \(O\)
    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:
    \(∀i < length ts.\) let (\,-,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot\)) = 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 \emptyset_i sb_i
    assume i-bound: i < length ts
    assume ts-i: ts ! i = (p_i,is_i,\emptyset_i,\emptyset_i,\emptyset_i,\emptyset_i,\emptyset_i,\emptyset_i,\emptyset_i)
    have all-acquired \(sb_i\) ∩ dom \(S\) − dom ?S' = \{\} ∧
        all-unshared \(sb_i\) ∩ dom ?S' − dom \(S\) = \{\}
    proof −
\textbf{from} ownership-distinct \textbf{of} 0 Suc \textit{i} ts-i t-i-bound \\
\textbf{have} dist: \((O \cup \text{all-acquired sb}) \cap (O_i \cup \text{all-acquired sb}_i) = \{\}\) \\
\hspace{1cm} \textbf{by} force \\

\textbf{from} dist unshared-acq-owned all-acquired-takeWhile \textbf{of} (Not \circ is-volatile-Write_{sb}sb) \textbf{have} all-acquired \textit{sb}_i \cap \text{dom } S - \text{dom } ?S' = \{\} \\
\hspace{1cm} \textbf{by} blast \\

\textbf{moreover} \\
\textbf{from} sharing-consis \textbf{of} Suc \textit{i} ts-i t-i-bound \\
\textbf{have} sharing-consistent \textit{S} \textit{O}_i \textit{sb}_i \\
\hspace{1cm} \textbf{by} force \\
\textbf{from} unshared-acquired-or-owned \textbf{OF} this \textbf{have} all-unshared \textit{sb}_i \subseteq \text{all-acquired \textit{sb}_i} \cup O_i . \\
\hspace{1cm} \textbf{with} dist fresh-owned \\
\textbf{have} all-unshared \textit{sb}_i \cap \text{dom } ?S' - \text{dom } S = \{\} \\
\hspace{1cm} \textbf{by} blast \\

\textbf{ultimately show} \textbf{?thesis} \textbf{by} simp \\
\hspace{1cm} \textit{qed} \\
\hspace{1cm} \{\} \\
\hspace{1cm} \textbf{thus} \textbf{?thesis} \\
\hspace{1.5cm} \textbf{by} (fastforce simp add: Let-def) \\
\hspace{1cm} \textit{qed} \\

\textbf{from} consis'.sharing-consis-preservation \textbf{OF} sep \\
\textbf{have} consis-ts: sharing-consis \textbf{?S'} ts. \\

\textbf{from} Cons.hyps \textbf{OF} dist' this leq' sim' \\
\textbf{obtain} consis-ts'"; \\
\hspace{1cm} sharing-consis (share-all-until-volatile-write ts \textbf{?S'}) ts'' \textbf{and} \\
\hspace{1.5cm} fresh: \text{dom } (share-all-until-volatile-write ts \textbf{?S'}) - \text{dom } ?S' \subseteq \\
\hspace{2cm} \bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,O,R)). O \circ \text{ set ts}) \textbf{and} \\
\hspace{1.5cm} unshared: \text{dom } ?S' - \text{dom } (share-all-until-volatile-write ts \textbf{?S'}) \subseteq \\
\hspace{2cm} \bigcup ((\lambda(\cdot,\cdot,\cdot,sb,\cdot,O,R)). \text{all-acquired sb} \cup O)^i \text{ set ts}) \\
\hspace{1cm} \textbf{by} auto \\

\textbf{from} sharing-consistent-append \textbf{OF} - - (takeWhile (Not \circ is-volatile-Write_{sb}sb) \textbf{have} consis-t' \textbf{?S' O'} sb' \\
\hspace{1cm} \textbf{by} (simp add: \textit{O'} \textit{sb'})
have fresh-dist: all-acquired sb' ∩ dom ?S' = dom (share-all-until-volatile-write ts ?S')

proof –
  have all-acquired sb' ∩ ∪ ((λ(,-,-,sb,-,O,-). all-acquired sb ∪ O)' set ts) = {}
  proof –
  { fix x
    assume x-sb': x ∈ all-acquired sb'
    assume x-ts: x ∈ ∪ ((λ(,-,-,sb,-,O,-). all-acquired sb ∪ O)' set ts)
    have False
    proof –
    from x-ts obtain i p i is i O i R i D i θ i sb i where
    i-bound: i < length ts and
    ts-i: ts!i = (p i, is i, θ i, sb i, D i, O i, R i) and
    x-in: x ∈ all-acquired sb i ∪ O i
    by (force simp add: in-set-conv-nth)
    from ownership-distinct [of 0 Suc i] ts-i t i-bound
    have dist: (O ∪ all-acquired sb) ∩ (O i ∪ all-acquired sb i) = {}
    by force
    with x-sb' x-in all-acquired-dropWhile [of (Not ◦ is-volatile-Write_{sb}) sb] show False
    by (auto simp add: sb')
    qed
  } thus ?thesis by blast
  qed
with unshared show ?thesis
  by blast
  qed

have unshared-dist: all-unshared sb' ∩ dom (share-all-until-volatile-write ts ?S') = dom ?S' = {}

proof –
  from unshared-acquired-or-owned [OF consis-t']
  have all-unshared sb' ⊆ all-acquired sb' ∪ O'.
  also
  from all-acquired-dropWhile [of (Not ◦ is-volatile-Write_{sb}) sb]
  acquired-all-acquired [of True takeWhile (Not ◦ is-volatile-Write_{sb}) sb O]
  all-acquired-takeWhile [of (Not ◦ is-volatile-Write_{sb}) sb]
  have all-acquired sb' ∪ O' ⊆ all-acquired sb ∪ O
  by (auto simp add: sb' O')
  finally
  have all-unshared sb' ⊆ (all-acquired sb ∪ O).

moreover

have (all-acquired sb ∪ O) ∩ ∪ ((λ(,-,-,,-,O,-). O)' set ts) = {}

proof –
{ fix x
assume x-sb': x ∈ all-acquired sb ∪ O
assume x-ts: x ∈ \( (λ(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,O,\cdot)). O) ^ t \) set ts
have False
proof
  from x-ts
  obtain i p_i i_s_i O_i R_i D_i \emptyset_i sb_i where
  i-bound: i < length ts and
  ts-i: ts!i = (p_i, i_s_i, \emptyset_i, sb_i, D_i, O_i, R_i) and
  x-in: x ∈ O_i
  by (force simp add: in-set-conv-nth)
  from ownership-distinct [of 0 Suc i] ts-i t i-bound
  have dist: \( (O ∪ all-acquired sb) \cap (O_i \cup all-acquired sb_i) = {} \)
  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-ts' fresh-dist unshared-dist]
have consis-ts: sharing-consistent (share-all-until-volatile-write ts \( ?S' \)) O' sb'.
note sharing-consis-Cons [OF consis-ts'' consis-ts, of p' is' \emptyset' D'']
moreover
  from fresh fresh-owned
  have dom (share-all-until-volatile-write ts \( ?S' \)) − dom S ⊆
     O ∪ \( (λ(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,O,\cdot)). O) ^ t \) set ts)
     by auto
  moreove
  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 \( ?S' \)) ⊆
     all-acquired sb ∪ O ∪ \( (λ(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,O,\cdot)). all-acquired sb ∪ O) ^ t \) 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 < length ts
assumes ts-i: ts!i = (p,i,s,\emptyset,sb,D,O,R)
shows sharing-consistent (share-all-until-volatile-write ts S)
(acquired True (takeWhile (Not ◦ is-volatile-Write sb) O))
(dropWhile (Not ◦ is-volatile-Write sb) sb)

proof –
define ts’ where ts’ ≜ map (λ(p, is, θ, sb, D, O, R).
  (p, is, θ, sb),
  dropWhile (Not ◦ is-volatile-Write sb) sb)
  O
  acquired True (takeWhile (Not ◦ is-volatile-Write sb) sb) O)
  ts
have leq: length ts’ = length ts
by (simp add: ts’-def)

have flush: ∀ i < length ts.
  (let (-, -, sb, -, O, -) = ts!i;
    (-, -, sb’, -, O’, -) = ts’!i
  in O’ = acquired True (takeWhile (Not ◦ is-volatile-Write sb) sb) O ∧
    sb’ = dropWhile (Not ◦ is-volatile-Write sb) sb)
by (auto simp add: 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 ts S) ts’ by simp
from i-bound leq ts-i sharing-consis [of i]
show ?thesis
by (force simp add: ts’-def)
qed

lemma restrict-map-UNIV [simp]: S [UNIV] = S
by (auto simp add: restrict-map-def)

lemma share-all-until-volatile-write-Read-commute:
shows ∀ S i. [i < length ls; ls!i=(p, Read volatile a t#is, θ, sb, D, O)]
⇒ share-all-until-volatile-write
  (ls[i := (p, is, θ(t→v), sb @ [Read sb volatile a t v]), D’, O)]) S =
  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 < length (l#ls))
note ith = ((l#ls)!i = (p, Read volatile a t#is, θ, sb, D, O))
show ?case
proof (cases i)
case 0
from \ith 0 have \( l = (p, \text{Read volatile a t\#is}, \varnothing, \text{sb}, D, O) \)
  by simp
thus \( ?\text{thesis} \)
  by (simp add: 0 share-append-Read sb del: fun-upd-apply )
next
  case (Suc \( n \))
  obtain \( p_1 \) \( \is_1 \) \( D_1 \) \( \varnothing_1 \) \( \sb_1 \) where
    \( l: l = (p_1, \is_1, \varnothing_1, \sb_1, D_1, O_1) \)
  by (cases \( l \))
  from \( \text{i-bound \ ith} \)
  have share-all-until-volatile-write
    \( (\text{ls}[n := (p, is, \varnothing(t\rightarrow v), \text{sb} \circ [\text{Read}_{\sb} \text{volatile a t v}], D', O)]) \)
    (share \( \text{(takeWhile (Not \circ \text{is-volatile-Write}_{\sb} \text{sb}) S)} = \) share-all-until-volatile-write \( \text{ls} \) \( \text{(share (takeWhile (Not \circ \text{is-volatile-Write}_{\sb} \text{sb}) S)} \)
      apply 
      apply (rule Cons.hyps)
      apply (auto simp add: Suc \( l \))
  done

  then
  show \( ?\text{thesis} \)
    by (simp add: Suc \( l \) del: fun-upd-apply )
qed

lemma share-all-until-volatile-write-Write-commute:
  shows \( \forall S i. \ [i < \text{length ls}; \text{ls}!i = (p, \text{Write volatile a t D f A L R W}\#is, \varnothing, \text{sb}, D, O) \] \)
    \( \text{⇒} \) share-all-until-volatile-write
    \( (\text{ls}![i := (p, is, \varnothing, \text{sb} \circ [\text{Write}_{\sb} \text{volatile a t f D W}] A L R, D', O)]) S = \) share-all-until-volatile-write \( \text{ls} \) \( S \)
proof (induct \( \text{ls} \))
  case Nil thus \( ?\text{case} \)
    by simp
next
  case (Cons \( l \) \( \text{ls} \))
  note \( \text{i-bound = \Series i < \text{length (} l \# \text{ls)}:} \)
  note \( \text{ith = \langle(} l \# \text{ls)!} i = (p, \text{Write volatile a t D f A L R W}\#is, \varnothing, \text{sb}, D, O) \rangle \)
  show \( ?\text{case} \)
proof (cases \( i \))
  case 0
  from \( \text{ith 0 have \( l: l = (p, \text{Write volatile a t D f A L R W}\#is, \varnothing, \text{sb}, D, O) \)} \)
    by simp
  thus \( ?\text{thesis} \)
    by (simp add: 0 share-append-Write sb del: fun-upd-apply )
next
  case (Suc \( n \))
  obtain \( p_1 \) \( \is_1 \) \( D_1 \) \( \varnothing_1 \) \( \sb_1 \) where
    \( l: l = (p_1, \is_1, \varnothing_1, \sb_1, D_1, O_1) \)
  by (cases \( l \))
  from \( \text{i-bound \ ith} \)
have share-all-until-volatile-write
(ls[n := (p,is, φ, sb @ [Write_{sb} volatile a t (f φ) A L R W',[D', O')]])
(share (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_l) S) =
share-all-until-volatile-write ls (share (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_l) 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-RMW-commute:
shows \( \forall S i. \ [i < \text{length} \ ls; \ ls![i]=(p,\text{RMW a t (D,f) cond ret A L R W',is,φ,[]},D,O)] \]
\( \Rightarrow \)
\( \text{share-all-until-volatile-write} \ (ls[i := (p',is, φ',[]),D',O']) \) S =
\( \text{share-all-until-volatile-write} \ ls \) S
proof (induct ls)
case Nil thus ?case
by simp
next
case (Cons l ls)
ote i-bound = \( \langle i < \text{length} \ (l#ls) \rangle \)
ote ith = \( \langle (l#ls)!i = (p,\text{RMW a t (D,f) cond ret A L R W',is,φ,[]},D,O) \rangle \)
show ?case
proof (cases i)
case 0 from ith 0 have l: l = (p,\text{RMW a t (D,f) cond ret A L R W',is,φ,[]},D,O)
by simp
thus ?thesis
by (simp add: 0 share-append-Write_{sb} del: fun-upd-apply )
next
case (Suc n)
obtain p_l is_l O_l D_l φ_l sb_l where l: l = (p_l,is_l,φ_l,sb_l,D_l,O_l)
by (cases l)
from i-bound ith
have share-all-until-volatile-write
(ls[n := (p',is, φ',[]),D',O'])
(share (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_l) S) =
share-all-until-volatile-write ls (share (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_l) 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-Fence-commute:
shows \( \forall S \ i. \ i < \text{length } ls; \ ls!i=(p,Fence#is,\emptyset,[]),D,O,R) \implies \)

\[
\text{share-all-until-volatile-write (ls[i := (p,is,\emptyset,[]),D,O,R'])} S = \text{share-all-until-volatile-write } ls \ S
\]

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,Fence#is,\emptyset,[]),D,O,R) \rangle
  show ?case
    proof (cases i)
      case 0
      from ith 0 have l: l = (p,Fence#is,\emptyset,[]),D,O,R)
        by simp
      thus ?thesis
        by (simp add: Suc l)
    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,\emptyset,l,SB,l,D,l,OO,l,AR)
        by (cases l)
      from i-bound ith
      have \[
      \text{share-all-until-volatile-write}
      (ls[n := (p,is,\emptyset,[]),D,O,R])
      (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_sb)} sb) S) =
      \text{share-all-until-volatile-write } ls \ (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_sb)} sb) S)
      \]
        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 unshared-share-in: \( \forall S. a \in \text{dom } S \Rightarrow a \notin \text{all-unshared } sb \Rightarrow a \in \text{dom (share sb S)} \)
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
show ?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 R \ominus A L)}
  by auto
from Cons.hyps [OF this a-sb]
show ?thesis
  by (clarsimp simp add: Write_{sb} True)
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
qed

lemma dom-eq-dom-share-eq: \( \forall S. \text{dom } S = \text{dom } S' \Rightarrow \text{dom (share sb S)} = \text{dom (share sb S')} \)
proof (induct sb)
case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
   case (Write x 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 x True)
   next
   case False with Cons.hyps [of S S'] Cons.prems Write x show ?thesis by auto
  qed
next
  case Read x with Cons.hyps [of S S'] Cons.prems show ?thesis by auto
next
  case Prog x with Cons.hyps [of S S'] Cons.prems show ?thesis by auto
next
  case (Ghost x 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 x)
  qed
qed
lemma share-union:
  \( \forall A B. [a \in \text{dom} (\text{share sb} (A \oplus z B)); a / \in \text{dom} A] \implies a \in \text{dom} (\text{share sb} (\text{Map.empty} \oplus z B)) \)
proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
   case (Write x volatile a' sop v A' L R W)
   show ?thesis
   proof (cases volatile)
    case True
    from Cons.prems
    obtain a-in: a \in \text{dom} (\text{share sb} ((A \oplus z B) \oplus_w R \ominus A' L)) and a-A: a / \in \text{dom} A
    by (clarsimp simp add: Write x True)
    have dom ((A \oplus z B) \oplus_w R \ominus A' L) \subset dom (A \oplus z (B \cup R - L))
  qed
qed


by auto
  from share-mono [OF this] a-in
  have a ∈ dom (share sb (A ⊕ (B ∪ R − L)))
by blast
  from Cons.hyps [OF this] a-A
  have a ∈ dom (share sb (Map.empty ⊕ (B ∪ R − L)))
by blast
  moreover
  have dom (Map.empty ⊕ B ∪ R − L) = dom ((Map.empty ⊕ 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 ⊕ B) ⊕ W R ⊖ A ⋃ L)) and a-A: a /∈ dom A
by (clarsimp simp add: Ghost sb)
have dom ((A ⊕ B) ⊕ W R ⊖ A ⋃ L) ⊆ dom (A ⊕ (B ∪ R − L))
by auto
from share-mono [OF this] a-in
have a ∈ dom (share sb (A ⊕ (B ∪ R − L)))
by blast
from Cons.hyps [OF this] a-A
have a ∈ dom (share sb (Map.empty ⊕ (B ∪ R − L)))
by blast
moreover
have dom (Map.empty ⊕ B ∪ R − L) = dom ((Map.empty ⊕ B) ⊕ 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)
qed
lemma share-unshared-in:
\[ \forall S. a \in \text{dom} \ (\text{share sb } S) \implies a \in \text{dom} \ (\text{share sb } \text{Map.empty}) \lor (a \in \text{dom} \ S \land a \notin \text{all-unshared sb}) \]

proof (induct sb)

\hspace{1em} case Nil thus ?case by simp

next
case (Cons x sb)
show ?case

proof (cases x)
case (\text{Write}_{sb} \ \text{volatile} \ a' \ \text{sop} \ v \ A \ L \ R \ W)
show ?thesis

proof (cases \text{volatile})
\hspace{1em} case True
\hspace{2em} note \text{volatile}=this
\hspace{2em} from Cons.prems
\hspace{2em} have a-in: \(a \in \text{dom} \ (\text{share sb } (S \oplus W R \ominus A L))\)

by (clarsimp simp add: \text{Write}_{sb} \ \text{volatile} \ True)
\hspace{2em} show ?thesis
\hspace{2em} proof (cases \(a \in \text{dom} \ S\))
\hspace{3em} case True
\hspace{4em} from Cons.hyps [OF a-in]
\hspace{4em} have \(a \in \text{dom} \ (\text{share sb } \text{Map.empty}) \lor a \in \text{dom} \ (S \oplus W R \ominus A L) \land a \notin \text{all-unshared sb}.\)
\hspace{4em} then show ?thesis
\hspace{2em} proof
\hspace{3em} assume \(a \in \text{dom} \ (\text{share sb } \text{Map.empty})\)
\hspace{3em} from share-mono-in [OF this]
\hspace{3em} have \(a \in \text{dom} \ (\text{share sb } (\text{Map.empty} \oplus W R \ominus A L))\) by auto
\hspace{3em} then show ?thesis
\hspace{4em} by (clarsimp simp add: \text{Write}_{sb} \ \text{volatile} \ True)
\hspace{2em} qed
\hspace{2em} next
\hspace{1em} case False
\hspace{2em} have \(\text{dom} \ (S \oplus W R \ominus A L) \subseteq \text{dom} \ (S \oplus W (R - L))\)
\hspace{3em} by auto
\hspace{2em} from share-mono [OF this] a-in
\hspace{2em} have \(a \in \text{dom} \ (\text{share sb } (S \oplus W (R - L)))\) by blast
\hspace{2em} from share-union [OF this False]
\hspace{2em} have \(a \in \text{dom} \ (\text{share sb } (\text{Map.empty} \oplus W (R - L)))\).
\hspace{2em} moreover
\hspace{3em} have \(\text{dom} \ (\text{Map.empty} \oplus W (R - L)) = \text{dom} \ (\text{Map.empty} \oplus W R \ominus A L)\)
\hspace{4em} by auto
\hspace{3em} note \text{dom-eq-dom-share-eq} [OF this, of sb]
\hspace{2em} ultimately
\hspace{2em} show ?thesis

qed
by (clarsimp simp add: Write sb True)
qeda

next
case False
with Cons show ?thesis
by (auto simp add: Write sb False)
qeda

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 ∈ dom (share sb (S ⊕ W R ⊖ A L))
  by (clarsimp simp add: Ghost sb)
show ?thesis
proof (cases a ∈ dom S)
  case True
  from Cons.hyps [OF a-in]
  have a ∈ dom (share sb Map.empty) ∨ a ∈ dom (S ⊕ W R ⊖ A L) ∧ a / ∈ all-unshared sb.
  then show ?thesis
  proof
    assume a ∈ dom (share sb Map.empty)
    from share-mono-in [OF this]
    have a ∈ dom (share sb (Map.empty ⊕ W R ⊖ A L)) by auto
    then show ?thesis
    proof
      by (clarsimp simp add: Ghost sb True)
    next
    assume a ∈ dom (S ⊕ W R ⊖ A L) ∧ a / ∈ all-unshared sb
    then obtain a / ∈ L a / ∈ all-unshared sb by auto
    then show ?thesis by (clarsimp simp add: Ghost sb True)
  qed
next
  case False
  have dom (S ⊕ W R ⊖ A L) ⊆ dom (S ⊕ 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: \( S \setminus R' \). dom R' = dom R ⇒
  dom (release sb S R 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

291
lemma dom-release-shared-eq: \( \forall S S' R. \ dom (\text{release sb } S' R) = \text{dom} (\text{release sb } S R) \)

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: \text{dom} (\text{augment-rels } S' R R) = \text{dom} (\text{augment-rels } S R R)
    by (rule dom-augment-rels-shared-eq)
  have \text{dom} (\text{release sb } (S' \cup R - L) (\text{augment-rels } S' R R)) =
    \text{dom} (\text{release sb } (S \cup R - L) (\text{augment-rels } S' R R))
    by (rule Cons.hyps)
  also have ... = \text{dom} (\text{release sb } (S \cup R - L) (\text{augment-rels } S R R))
    by (rule dom-release-rels-eq [OF dr])
  finally show ?thesis
    by (clarsimp simp add: Ghost\(_{sb} \))
  qed
qend

lemma share-other-untouched:
\( \forall O S. \ \text{sharing-consistent } S O \ \text{sb} \Rightarrow a \notin O \cup \text{all-acquired sb} \Rightarrow \text{share sb } S a = S a \)

proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write\(_{sb} \) volatile a' sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True
    qed
  qed

from Cons.prems obtain
A-shared-owns: \( A \subseteq \text{dom } S \cup O \ \text{and } \text{L-A}: L \subseteq A \ \text{and } \text{A-R}: A \cap R = \{\} \ \text{and } \text{R-owns: } R \subseteq O \ \text{and } \text{consis'}: \text{sharing-consistent } (S \oplus R \ominus A L) (O \cup A - R) \ \text{sb} \ \text{and } \text{a-owns: a} \notin O \ \text{and } \text{a-A}: a \notin A \ \text{and } \text{a-sb: a} \notin \text{all-acquired sb} \)

by (simp add: Write\(_{sb} \ True \))

from a-owns a-A a-sb
have a \notin O \cup A - R \cup \text{all-acquired sb}
  by auto

292
from Cons.hyps [OF consis’ this]
have share sb \((S \oplus_W R \ominus_A L) \ a = (S \oplus_W R \ominus_A L) \ a\).
moreover have \((S \oplus_W R \ominus_A L) \ a = S \ a\)
using L-A A-R R-owns a-owns a-A
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
ultimately show ?thesis
by (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)
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 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 \((S \oplus_W R \ominus_A L) (O \cup A \ominus R) \ sb\) 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 )

from a-owns a-A a-sb
have a \notin O \cup A \ominus R \cup all-acquired \sb
by auto
from Cons.hyps [OF consis’ this]
have share sb \((S \oplus_W R \ominus_A L) \ a = (S \oplus_W R \ominus_A L) \ a\).
moreover have \((S \oplus_W R \ominus_A L) \ a = S \ a\)
using L-A A-R R-owns a-owns a-A
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
ultimately show ?thesis
by (simp add: Ghost\sb)
qed

lemma shared-owned: \(\land O \ S. \ sharing-consistent \ S \ O \ sb \ \Rightarrow \ a \notin dom \ S \ \Rightarrow \ a \in dom \ (share \ sb \ S) \ \Rightarrow \ a \in O \cup all-acquired \ sb\)

proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write\textsubscript{sb} volatile a′ sop v A L R W)
show ?thesis
proof (cases volatile)
  case True

  from Cons.prems obtain
  A-shared-owns: A ⊆ dom \( \mathcal{S} \cup \mathcal{O} \) and L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns:
  R ⊆ \( \mathcal{O} \) and
  consis′: sharing-consistent (\( \mathcal{S} \oplus_{\textbf{W}} R \ominus_{\text{A}} L \)) (\( \mathcal{O} \cup A − R \)) sb and
  a-notin: a /∈ dom \( \mathcal{S} \) and a-in: a ∈ dom (share sb (\( \mathcal{S} \oplus_{\textbf{W}} R \ominus_{\text{A}} L \))
by ( simp add: Write\textsubscript{sb} True )

  show ?thesis
  proof (cases a ∈ \( \mathcal{O} \))
    case True thus ?thesis by auto
  next
    case False
    with a-notin R-owns A-shared-owns L-A A-R have a /∈ dom (\( \mathcal{S} \oplus_{\textbf{W}} R \ominus_{\text{A}} L \))
    by (auto)
    from Cons.hyps [OF consis′ this a-in]
    show ?thesis
    by (auto simp add: Write\textsubscript{sb} True)
  qed

  next
    case False with Cons show ?thesis
    by (auto simp add: Write\textsubscript{sb} False)
  qed

next
  case Read\textsubscript{sb} with Cons
  show ?thesis
  by (auto)

next
  case Prog\textsubscript{sb} with Cons
  show ?thesis
  by (auto)

next
  case (Ghost\textsubscript{sb} A L R W)
  from Cons.prems obtain
  A-shared-owns: A ⊆ dom \( \mathcal{S} \cup \mathcal{O} \) and L-A: L ⊆ A and A-R: A ∩ R = {} and R-owns:
  R ⊆ \( \mathcal{O} \) and
  consis′: sharing-consistent (\( \mathcal{S} \oplus_{\textbf{W}} R \ominus_{\text{A}} L \)) (\( \mathcal{O} \cup A − R \)) sb and
  a-notin: a /∈ dom \( \mathcal{S} \) and a-in: a ∈ dom (share sb (\( \mathcal{S} \oplus_{\textbf{W}} R \ominus_{\text{A}} L \))
by (simp add: Ghost\textsubscript{sb})

  show ?thesis
  proof (cases a ∈ \( \mathcal{O} \))
    case True thus ?thesis by auto
  next
    case False
    with a-notin R-owns A-shared-owns L-A A-R have a /∈ dom (\( \mathcal{S} \oplus_{\textbf{W}} R \ominus_{\text{A}} L \))

294
by (auto)
from Cons.hyps [OF cons' this a-in]
show ?thesis
  by (auto simp add: Ghost)
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]
  by auto

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 D OR. i < length ts → ts!i = (p, is, θ, sb, δ, 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 p t is t D t θ t sb t where
  t: t=(p, is, θ, sb, δ, O, R)
    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, δ, O, R) →
    a /∈ O ∪ all-acquired sb and
  a-notin: a /∈ O ∪ all-acquired sb
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
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.

295
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-Writesb) sbt) S)

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent S Ot sbt.
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent S Ot (takeWhile (Not ◦ is-volatile-Writesb) sbt).
from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom ?S' − dom S ⊆ Ot.
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) sbt) ∪ Ot
  by simp

have sep:
  ∀ i < length ts. let (r, r, sb', 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 v_i sb_i
    assume i-bound: i < length ts
    assume ts-i: ts ! i = (p_i,is_i,v_i,rb_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 i-bound
    have dist: (Ot ∪ all-acquired sb_i) ∩ (Ot ∪ all-acquired sb_i) = {}
    by force
    from dist unshared-acq-owned all-acquired-takeWhile [of (Not ◦ is-volatile-Writesb) 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 Ot sb_i
  by force
from unshared-acquired-or-owned [OF this]
have all-unshared sb_i ⊆ all-acquired sb_i ∪ Ot,
with dist fresh-owned
have all-unshared sb₁ ∩ 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 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 ◦ is-volatile-Write sbₜ) sbₜ) S a .
moreover
from share-other-untouched [OF consis-sb] a-notin
   all-acquired-append [of (takeWhile (Not ◦ is-volatile-Write sbₜ) sbₜ) (dropWhile (Not ◦ is-volatile-Write sbₜ) sbₜ)]
have share (takeWhile (Not ◦ is-volatile-Write sbₜ) sbₜ) S a = S a
by auto
ultimately
show ?case
   by (simp add: t)
qed

lemma share-shared-eq: \(\forall S' S. S' a = S a \implies \) share sb S' a = 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' ⊕_W R ⊕ₐ L) a = (S ⊕_W R ⊕ₐ L) a
       using eq by (auto simp add: augment-shared-def restrict-shared-def)
       from Cons.hyps [of (S' ⊕_W R ⊕ₐ L) (S ⊕_W R ⊕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 \text{S}' \text{S}] Cons.prems show \?thesis
    by (auto simp add: Read\sb)
next
case Prog\sb
  with Cons.hyps [of \text{S}' \text{S}] Cons.prems show \?thesis
    by (auto simp add: Read\sb)
next
case (Ghost\sb A L R W)
  have \((\text{S}' \oplus W R \ominus A L) a = (\text{S} \oplus W R \ominus A L) a\)
    using eq by (auto simp add: augment-shared-def restrict-shared-def)
  from Cons.hyps [of \((\text{S}' \oplus W R \ominus A L) (\text{S} \oplus W R \ominus A L), \text{OF this}\)]
    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 \text{S} ts
  assumes i-bound: \(i < \text{length ts}\)
  assumes ts-i: ts!i = \((p,\text{is},\theta,\text{sb},D,O,R)\)
  assumes a-owned: \(a \in O \cup \text{all-acquired sb}\)
  shows share-all-until-volatile-write ts \text{S} a = share (takeWhile (Not ◦ is-volatile-Write\sb) \text{S} a)
  using dist consis i-bound ts-i
proof (induct ts arbitrary: \text{S} i)
  case Nil thus \?case by simp
next
case (Cons t ts)
  obtain \(p_t \text{is}_t \text{R}_t \text{D}_t \theta_t \text{sb}_t\) where
    \(t: t=(p_t,\text{is}_t,\theta_t,\text{sb}_t,\text{D}_t,\text{O}_t,\text{R}_t)\)
    by (cases t)
  have dist: ownership-distinct \((t\#ts)\) by fact
  then interpret ownership-distinct \(t\#ts\).
  have consis: sharing-consis \text{S} \((t\#ts)\) by fact
  then interpret sharing-consis \text{S} \(t\#ts\).
from ownership-distinct-tl \[\text{OF dist}\]
have dist': ownership-distinct ts.
from sharing-consis-tl \[\text{OF consis}\]
have consis': sharing-consis \text{S} ts.
then interpret consis': sharing-consis \text{S} ts.
let \(?\text{S}' = (\text{share (takeWhile (Not ◦ is-volatile-Write}_{\text{sb}}) \text{sb}) \text{S})\)
from sharing-consis [of 0, simplified, OF t]
have sharing-consistent $S \cdot O_i \cdot sb_i$.
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent $S \cdot O_i$ (takeWhile (Not o is-volatile-Write$sb_i$) sb$_i$).
from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom $?S' - dom S \subseteq O_i$.
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 (takeWhile (Not o is-volatile-Write$sb_i$) sb$_i$) $\cup O_i$
  by simp

have sep:
  $\forall i <$ length ts. let $(r_r_r, sb'_r, r_r_r) = ts!i$ in
  all-acquired sb$'_i$ $\cap$ dom $S - dom ?S' = \{}$ $\wedge$
  all-unshared sb$'_i$ $\cap$ dom $?S' - dom S = \{}$
proof
  { 
    fix i p$_i$ is$_i$ R$_i$ D$_i$ $\theta_i$ sb$_i$
    assume i-bound: $i <$ length ts
    assume ts-i: ts ! i = $(p_i, is_i, \theta_i, sb_i, D_i, O_i, R_i)$
    have all-acquired sb$_i$ $\cap$ dom $S - dom ?S' = \{}$ $\wedge$
      all-unshared sb$_i$ $\cap$ dom $?S' - dom S = \{}$
    proof
      from ownership-distinct [of 0 Suc i] ts-i t i-bound
      have dist: $(O_i \cup$ all-acquired sb$_i$) $\cap$ $(O_i \cup$ all-acquired sb$_i$) = \{}
        by force
      more-over
    from dist unshared-acq-owned all-acquired-takeWhile [of (Not o is-volatile-Write$sb_i$) sb$_i$]
      have all-acquired sb$_i$ $\cap$ dom $S - dom ?S' = \{}$
        by blast
      moreover
    from sharing-consis [of Suc i] ts-i t i-bound
      have sharing-consistent $S \cdot O_i$ 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$_i$ $\cap$ 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 this]

have consis-shared'; sharing-consis ?S' ts.

have aargh: (Not ◦ is-volatile-Write_{sb}) = (λa. ¬ is-volatile-Write_{sb} 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
    qed
  }

  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 t' t
    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-Write_{sb}) 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_t) (dropWhile (Not ◦ is-volatile-Write_{sb}) sb_t)]
    have ?S' a = S a
      by auto
  from share-shared-eq [of ?S' a S,OF this ]

300
have share (takeWhile (Not ◦ is-volatile-Write sb) S) ?S' a = share (takeWhile (Not ◦ is-volatile-Write 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) S) (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-unti-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-unti-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:
\[ S \subseteq O. \] [sharing-consistent \( S \subseteq O \) \( a \in O \); \( a \notin \) all-unshared \( sb \)]
\[ \implies a \in \text{dom} (\text{share} \ sb \ S) \cup \text{acquired} \ True \ sb \ O \]

proof (induct \( sb \))
case Nil thus ?case by auto
next
case (Cons \( x \) \( sb \))
show ?case
proof (cases \( x \))
case (Write\( _{ab} \) volatile \( a' \) sop \( v \) \( A \ L \ R \ W \))
show ?thesis
proof (cases volatile)
case True
note volatile=\( \text{this} \)
from Cons.prems obtain
a-owns: \( a \in O \) and A-shared-onws: \( A \subseteq \text{dom} S \cup O \) and
a-L: \( a \notin L \) and a-unsh: \( a \notin \) all-unshared \( sb \) 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\( _{ab} \) volatile)
have \( a \in \text{dom} (\text{share} \ sb \ (S \oplus W R \ominus A L)) \cup \text{acquired} True \ sb \ (O \cup A - R) \)
proof (cases \( a \in R \))
case True
with a-L have \( a \in \text{dom} (S \oplus W R \ominus A L) \)
by auto
from unshared-share-in [OF this a-unsh]
show ?thesis by blast
next
case False
hence \( a \in O \cup A - R \)
using a-owns
by auto
from Cons.hyps [OF consis’ this a-unsh]
show ?thesis .
qed
then
show ?thesis
by (clarsimp simp add: Write\( _{ab} \) volatile)
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

302
case Progsb
with Cons show \(?\)thesis
  by (auto simp add: Readsb)

next
case (Ghostsb A L R W)
from Cons.prems obtain
  a-owns: a ∈ \(O\) and A-shared-owns: 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 (\(S ⊕_W R ⊖_A L\)) (\(O ∪ A - R\)) sb
by (clarsimp simp add: Ghostsb)
have a ∈ dom (share sb (\(S ⊕_W R ⊖_A L\)) ∪ acquired True sb (\(O ∪ A - R\))
proof (cases a ∈ R)
case True
  with a-L have a ∈ dom (\(S ⊕_W R ⊖_A L\))
  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: Ghostsb)
  qed
  qed

lemma shared-share-acquired: \(\{\} \cup S \cup O\). sharing-consistent \(S \cup O\) sb \(\Rightarrow\)
  a ∈ dom \(S\) \(\Rightarrow\) a ∈ dom (share sb \(S\)) ∪ 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 ∈ 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

303
proof (cases a ∈ L)
case False with a-shared
have a ∈ dom \((S ⊕_W R ⊕_A L)\)
  by auto
from Cons.hyps [OF conis\' 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 - conis\', where S'=(S ⊕_W R)]
have sharing-consistent \((S ⊕_W R) (O \cup A - R)\) sb
  by auto
from Cons.hyps [OF this] a-shared
have hyp: a ∈ dom (share sb \((S ⊕_W R)\)) ∪ acquired True sb \((O \cup A - R)\)
  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 \((O \cup A - R)\)
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 \notin all-unshared sb
  hence a-unsh: a \notin all-unshared sb by blast
from a-A A-R have a ∈ O \cup A - R
  by auto
from owns-unshared-share-acquired [OF conis\' 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\sb)
  qed
next
case Read\sb
  with Cons show ?thesis
    by (auto simp add: Read\sb)
next

304
case Prog\textsubscript{sb}
  with Cons show \textit{?thesis}
  by (auto simp add: Read\textsubscript{sb})

next
  case (Ghost\textsubscript{sb} A L R W)
  from Cons.prems obtain
    a-shared: \( a \in \text{dom} \ S \) 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 \((S \oplus W R \ominus A L) (O \cup A - R)\) sb
    by (clarsimp simp add: Ghost\textsubscript{sb})
  show \textit{?thesis}
  proof (cases \( a \in L \))
    case False with a-shared
    have \( a \in \text{dom} \ (S \oplus W R \ominus A L) \)
      by auto
    from Cons.hyps [OF consis' this]
    show \textit{?thesis}
      by (clarsimp simp add: Ghost\textsubscript{sb})
    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: \( a \in \text{dom} \ (\text{share sb} (S \oplus W R)) \cup \text{acquired True sb} (O \cup A - R) \)
      by auto
      {
        assume \( a \in \text{dom} \ (\text{share sb} (S \oplus W R)) \)
        from share-unshared-in [OF this]
        have \( a \in \text{dom} \ (\text{share sb} (S \oplus W R \ominus A L)) \cup \text{acquired True sb} (O \cup A - R) \)
          proof
            assume \( a \in \text{dom} \ (\text{share sb Map.empty}) \)
            from share-mono-in [OF this]
            have \( a \in \text{dom} \ (\text{share sb} (S \oplus W R \ominus A L)) \)
              by auto
            thus \textit{?thesis} by blast
          next
            assume \( a \in \text{dom} \ (S \oplus W R) \) \& \( a \notin \text{all-unshared sb} \)
            hence a-unsh: \( a \notin \text{all-unshared sb} \) by blast
            from a-A A-R have \( a \in O \cup A - R \)
              by auto
            from owns-unshared-share-acquired [OF consis' this a-unsh]
            show \textit{?thesis}.
              qed
          } 
        with hyp show \textit{?thesis}
          by (auto simp add: Ghost\textsubscript{sb})
      qed

305
lemma dom-release-takeWhile:
\[ S \quad R. \quad \]
\[ \text{dom} \ (\text{release} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \ S \ R) = \]
\[ \text{dom} \ R \cup \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \]
apply (induct sb)
apply (clarsimp)
subgoal for a sb S 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 \ (\text{share-all-until-volatile-write} \ ts S)
shows \[ \exists i < \text{length} \ ts. \]
\[ \text{let} \ (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \text{ts}!i \]
in a \in \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb)
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 i_s t_O t_R t_D t_\theta t_sb by cases t
by (cases t)

let ?take = (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_t)
from t Cons.prems
have a-in: a \in dom \ (\text{share-all-until-volatile-write} \ ts \ (\text{share} ?\text{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 $S \subseteq O_t$.
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 ?take $\cup O_t$
by simp

have sep:
$\forall i < \text{length ts.} \text{let } (-,-,-,sb'_i,-,-) = \text{ts}!i \text{ in}$
all-acquired $sb'_i$ $\cap$ dom $S −$ dom $?S' = \{\} \wedge$
all-unshared $sb'_i$ $\cap$ dom $?S' −$ dom $S = \{\}$

proof –
{
fix $i$ $p_i$ $is_i$ $R_i$ $D_i$ $\vartheta_i$ $sb_i$
assume i-bound: $i < \text{length ts}$
assume ts-i: $\text{ts}!i = (p_i,is_i,\vartheta_i,sb_i,D_i,O_i,R_i)$
have all-acquired $sb_i$ $\cap$ dom $S −$ dom $?S' = \{\} \wedge$
all-unshared $sb_i$ $\cap$ dom $?S' −$ dom $S = \{\}$
proof –
from ownership-distinct [of 0 Suc $i$] ts-i i-bound
have 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 $\circ\,$ is-volatile-Write$_{sb}$) $sb_i$]
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 $S$ $O_i$ $sb_i$
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 $sb_i$ $\cap$ 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 this]

have consis-shared': sharing-consis ?S' ts.

have aargh: (Not ◦ is-volatile-Write_{sb}) = (λa. ¬ is-volatile-Write_{sb} a)
by (rule ext) auto

show ?case
proof (cases a ∈ 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 ∉ dom ?S'
proof
  assume a ∈ 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, θ, sb, D, O, R) = ts!i
    in a ∈ all-shared (takeWhile (Not ◦ is-volatile-Write_{sb}) 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

lemma all-shared-share-acquired: \( \forall S O. \) sharing-consistent \( S O sb \implies \)
a ∈ all-shared sb \( \implies a ∈ \) dom (share sb \( S \)) \cup 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 ∪ all-shared sb 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 ∈ 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 ∪ all-shared sb 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 ∈ 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 ∈ 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: Ghost sb)
qed
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, is, φ, sb, D, O, R)
  assumes a-in: a ∈ dom (share-all-until-volatile-write ts S)
  shows a ∈ dom (share sb S) ∨ a ∈ acquired True sb O ∨
  (∃ j < length ts. j ≠ i ∧
   (let (p, is, φ, sb j, D, O, R) = ts!j
    in a ∈ all-shared (takeWhile (Not ◦ is-volatile-Write sb j) 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 o is-volatile-Write sb) sb
  let ?drop-sb = dropWhile (Not o is-volatile-Write sb) sb

  show ?thesis
  proof (cases a ∈ dom S)
    case True
    from shared-share-acquired [OF consis-sb True]
    have a ∈ dom (share sb S) ∪ 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 (p, is, φ, sb j, D, O, R) = 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:
∀A. a ∈ acquired True sb A ⇒ a ∈ acquired True sb {} ∨ (a ∈ A ∧ a /∈ 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.
then show ?thesis
proof

assume $a \in \text{acquired True sb }$ \{\}
from \text{acquired-mono-in [OF this]}
have $a \in \text{acquired True sb } (A' - R)$ \text{by auto}
then show ?thesis
by (clarsimp simp add: \text{Write}_{sb} \text{ volatile True})
next
assume $a \in A \cup A' - R \land a \notin \text{all-shared sb}$
then obtain $a \notin R \land a \notin \text{all-shared sb}$
by blast
then show ?thesis by (clarsimp simp add: \text{Write}_{sb} \text{ volatile True})
qed
next
case False
have $(A \cup A' - R) \subseteq A \cup (A' - R)$
by blast
from \text{acquired-mono [OF this] a-in}
have $a \in \text{acquired True sb } (A \cup (A' - R))$ \text{by blast}
from \text{acquired-union-notin-first [OF this False]}
have $a \in \text{acquired True sb } (A' - R)$.
then show ?thesis
by (clarsimp simp add: \text{Write}_{sb} \text{ True})
qed
next
with Cons show ?thesis
by (auto simp add: \text{Write}_{sb} \text{ False})
next
case Read_{sb}
with Cons show ?thesis
by (auto simp add: \text{Read}_{sb})
next
case Prog_{sb}
with Cons show ?thesis
by (auto simp add: \text{Read}_{sb})
next
case (\text{Ghost}_{sb} \ A' \ L \ R \ W)
from Cons.prems
have a-in: $a \in \text{acquired True sb } (A \cup A' - R)$
by (clarsimp simp add: \text{Ghost}_{sb})
show ?thesis
proof (cases $a \in A$)
case True
from Cons.hyps [OF a-in]
have $a \in \text{acquired True sb }$ \{\} \or $a \in A \cup A' - R \land a \notin \text{all-shared sb}.$
then show ?thesis
proof
assume $a \in \text{acquired True sb }$ \{\}
from \text{acquired-mono-in [OF this]}
have $a \in \text{acquired True sb } (A' - R)$ \text{by auto}
then show ?thesis
by (clarsimp simp add: Ghost\textsubscript{sb} 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: Ghost\textsubscript{sb} 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: Ghost\textsubscript{sb})
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\textsubscript{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: Write\textsubscript{sb} 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\textsubscript{sb} True)
qed
next
case False
with Cons show ?thesis
by (auto simp add: Write\textsubscript{sb} False)
qed
next
case Read\textsubscript{sb}
  with Cons show \textit{thesis}
    by (auto simp add: Read\textsubscript{sb})
next
case Prog\textsubscript{sb}
  with Cons show \textit{thesis}
    by (auto simp add: Read\textsubscript{sb})
next
case Ghost\textsubscript{sb}
  with Cons show \textit{thesis}
    by (auto simp add: Ghost\textsubscript{sb})
qed

lemma owned-share-acquired: \( \forall S \mathcal{O}. \) sharing-consistent \( S \mathcal{O} \) \( \text{sb} \implies a \in \mathcal{O} \implies a \in \text{dom} (\text{share \textsubscript{sb} } S) \cup \text{acquired True } \text{sb} \mathcal{O} \)
proof (induct \text{sb})
  case Nil thus \textit{case} by auto
next
case (Cons \textit{x} \text{sb})
  show \textit{case}
    proof (case \textit{x})
      case (Write\textsubscript{sb} volatile \textit{a'} sop v A L R W)
      show \textit{thesis}
        proof (cases \textit{volatile})
          case True
          note volatile=this
          from Cons\textsubscript{sb} prems obtain
            a-owned: \( a \in \mathcal{O} \) and \( A\text{-\textsubscript{shared-owns}}: A \subseteq \text{dom } S \cup \mathcal{O} \) and
            L-A: \( L \subseteq A \) and \( A\text{-\textsubscript{R}}: A \cap R = \{\} \) and \( R\text{-\textsubscript{owns}}: R \subseteq \mathcal{O} \) and
            consis': sharing-consistent \((S \oplus _W R \ominus_A L) \ (O \cup A - R) \) \( \text{sb} \)
          by (clarsimp simp add: Write\textsubscript{sb} True)
          show \textit{thesis}
            proof (cases \( a \in R \))
          case False with a-owned
          have \( a \in O \cup A - R \)
            by auto
          from Cons\textsubscript{sb} hyps \{OF consis' \text{this}\}
          show \textit{thesis}
            proof (clarsimp simp add: Write\textsubscript{sb} volatile)
              next
case True
from True L-A-R have \( a \in \text{dom } (S \oplus _W R \ominus_A L) \)
  by auto
from shared-share-acquired \{OF consis' \text{this}\}
show \textit{thesis}
  proof (clarsimp simp add: Write\textsubscript{sb} volatile True)
qed
next

314
case False
with Cons
show ?thesis
by (auto simp add: Write)
qed
next
case Read sb
with Cons show ?thesis
by (auto simp add: Read)
next
case Prog sb
with Cons show ?thesis
by (auto simp add: Read)
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)
show ?thesis
proof (cases a ∈ R)
case False with a-owned
have a ∈ O ∪ A − R
by auto
from shared-share-acquired [OF consis' this]
show ?thesis
by (clarsimp simp add: Ghost)
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 True)
qed
qed

case outstanding-refs-non-volatile-Read sb all-acquired:
Λm S O pending-write.
[reads-consistent pending-write O m sb;non-volatile-owned-or-read-only pending-write
S O sb;
a ∈ outstanding-refs is-non-volatile-Read sb]
⇒ a ∈ O ∨ a ∈ all-acquired sb ∨
a ∈ 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  
  note volatile=this  
  from Cons.prems obtain  
  non-vo: non-volatile-owned-or-read-only True (S ⊕ W R ⊕ A L)  
  (O ∪ A − R) sb and  
  out-vol: outstanding-refs is-volatile-Read sb sb = {} and  
  out: a ∈ outstanding-refs is-non-volatile-Read sb sb  
  by (clarsimp simp add: Write sb True)  
  show ?thesis  
  proof (cases a ∈ O)  
  case True  
  show ?thesis  
  proof (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 ∈ acquired-reads True sb (O ∪ A − R) ∨  
  a ∈ read-only-reads (O ∪ A − R) sb  
  by auto  
  with acquired-reads-all-acquired [of True sb (O ∪ A − R)]  
  show ?thesis  
  proof (auto simp add: Write sb volatile)  
  qed  
  next  
  case False  
  with Cons 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 Prog sb  
  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
lemma outstanding-refs-non-volatile-Read\_sb-all-acquired-dropWhile:
assumes consis: reads-consistent pending-write \(\mathcal{O}\) m sb
assumes nvo: non-volatile-owned-or-read-only pending-write \(\mathcal{S} \mathcal{O}\) sb
assumes out: \(a \in\) outstanding-refs is-non-volatile-Read\_sb (dropWhile (Not \(\circ\) is-volatile-Write\_sb) sb)
shows \(a \in \mathcal{O} \lor a \in\) all-acquired sb \(\lor\)
\(a \in\) read-only-reads \(\mathcal{O}\) sb
using outstanding-refs-append [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 [OF consis nvo, of a] out
by (auto)

lemma share-commute:
\[\bigwedge L R S O. [\text{sharing-consistent } S O sb];
\text{all-shared sb} \cap L = \{\}; \text{all-shared sb} \cap A = \{\}; \text{all-acquired sb} \cap R = \{\};
\text{all-unshared sb} \cap R = \{\}; \text{all-shared sb} \cap R = \{\}] \implies
\begin{align*}
\text{(share sb } (S \oplus W R \ominus A L)) &= \\
\text{(share sb } S) \oplus W R \ominus A L
\end{align*}
proof (induct sb)
\hspace{1em} case Nil thus ?case by simp
next
\hspace{1em} case (Cons x sb)
\hspace{1em} show ?case
\hspace{1em} proof (cases x)
\hspace{1em} \hspace{1em} case (Write\_sb volatile a sop v A' L' R' W')
\hspace{1em} \hspace{1em} show ?thesis
\hspace{1em} \hspace{1em} proof (cases volatile)
\hspace{1em} \hspace{1em} \hspace{1em} case True
\hspace{1em} \hspace{1em} \hspace{1em} note volatile=this from Cons.prems obtain
\hspace{1em} \hspace{1em} \hspace{1em} L-prop: (R' \cup \text{all-shared sb}) \cap L = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} A-prop: (R' \cup \text{all-shared sb}) \cap A = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} R-acq-prop: (A' \cup \text{all-acquired sb}) \cap R = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} R-prop: (L' \cup \text{all-unshared sb}) \cap R = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} R-prop-sh: (R' \cup \text{all-shared sb}) \cap R = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} A'-shared-owns: A' \subseteq \text{dom } S \cup O \text{ and } L'\subseteq A' \text{ and } A'-R': A' \cap R' = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} R'-owns: R' \subseteq O \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} consis': \text{sharing-consistent } (S \oplus_W R' \ominus_{A'} L' \cup O \cup A' - R') sb
\hspace{1em} \hspace{1em} \hspace{1em} by (clarsimp simp add: Write\_sb volatile)
\hspace{1em} \hspace{1em} \hspace{1em} from L-prop obtain R'-L: R' \cap L = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} acq-L: \text{all-shared sb} \cap L = \{\}
\hspace{1em} \hspace{1em} \hspace{1em} by blast
\hspace{1em} \hspace{1em} \hspace{1em} from A-prop obtain R'-A: R' \cap A = \{\} \text{ and}
\hspace{1em} \hspace{1em} \hspace{1em} acq-A: \text{all-shared sb} \cap A = \{\}

317
by blast
from R-acq-prop obtain $A' \cap R = \{\}$ and acq-R: all-acquired sb $\cap R = \{\}$
by blast
from R-prop obtain $L' \cap R = \{\}$ and unsh-R: all-unshared sb $\cap R = \{\}$
by blast
from R-prop-sh obtain $R' \cap R = \{\}$ and sh-R: all-shared sb $\cap R = \{\}$
by blast

from Cons.hyps [OF consis' acq-L acq-A acq-R unsh-R sh-R ]

have share sb $((S \oplus W \ R \ominus A \ L) \oplus W R \ominus A L) = share sb ((S \oplus W \ R' \ominus A' L') \oplus W R \ominus A L$.

moreover

from $R' L' R' A' R$

have $((S \oplus W R \ominus A L) \oplus W R' \ominus A' L') = ((S \oplus W R' \ominus A' L') \oplus W R \ominus A L$)

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 share sb $((S \oplus W R \ominus A L) \oplus W R' \ominus A' L') = share sb ((S \oplus W R' \ominus A' L') \oplus W R \ominus A L$.

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: $(R' \cup all-shared sb) \cap L = \{\}$ 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: $A' \subseteq \text{dom } S \cup \mathcal{O}$ and $L' \subseteq A'$ and $A' \cap R' = \{\}$ and
R'-owns: $R' \subseteq \mathcal{O}$ and
consis': sharing-consistent $(S \oplus W R' \ominus A' L') (\mathcal{O} \cup A' - R')$ sb
by (clarsimp simp add: Ghost sb)

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 A' A R
have ((S ⊕ W R ⊕ A L) ⊕ W' R' ⊕ A' L') = ((S ⊕ W' R' ⊕ A L) ⊕ W R ⊕ 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 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: Ghost sb)
qed
qed

lemma share-all-until-volatile-write-commute:
∀ S R L. [ownership-distinct ts; sharing-consis S ts;
  ∀ i p is O R D ⊥ sb. i < length ts → tlsi=(p,i,s,θ,si,db,D,O,R) →
    all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ L = {};
  ∀ i p is O R D ⊥ sb. i < length ts → tlsi=(p,i,s,θ,si,db,D,O,R) →
    all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ A = {};
  ∀ i p is O R D ⊥ sb. i < length ts → tlsi=(p,i,s,θ,si,db,D,O,R) →
    all-acquired (takeWhile (Not o is-volatile-Write sb) sb) ∩ R = {};
  ∀ i p is O R D ⊥ sb. i < length ts → tlsi=(p,i,s,θ,si,db,D,O,R) →
    all-unshared (takeWhile (Not o is-volatile-Write sb) sb) ∩ R = {};
∀ i p is O R D ⊥ sb. i < length ts → tlsi=(p,i,s,θ,si,db,D,O,R) →

319
all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ R = {}]

share-all-until-volatile-write ts S ⊕ W R ⊖ A L = 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 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.

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,∅,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,∅,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,∅,sb,D,O,R) →

all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ A = {} by fact

hence A-prop': ∀ i p is O R D ⊥ sb. i < length (ts) → (ts)[i]=(p,is,∅,sb,D,O,R) →

all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ A = {} by force

have R-prop-acq: ∀ i p is O R D ⊥ sb. i < length (t#ts) → (t#ts)[i]=(p,is,∅,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,∅,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,∅,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,∅,sb,D,O,R) →

all-unshared (takeWhile (Not o is-volatile-Write sb) sb) ∩ R = {} by force

have R-prop-sh: ∀ i p is O R D ⊥ sb. i < length (t#ts) → (t#ts)[i]=(p,is,∅,sb,D,O,R) →

all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ R = {} by fact

hence R-prop-sh': ∀ i p is O R D ⊥ sb. i < length (ts) → (ts)[i]=(p,is,∅,sb,D,O,R) →

all-shared (takeWhile (Not o is-volatile-Write sb) sb) ∩ R = {}
by force

from ownership-distinct-tl [OF dist]
have dist': ownership-distinct ts.

from sharing-consis-tl [OF consis]
have consis': sharing-consis $\mathcal{S}$ ts.
then
interpret consis': sharing-consis $\mathcal{S}$ ts.

from L-prop [rule-format, of 0 p is $\emptyset$ sb $\mathcal{D} \mathcal{O}$] t
have sh-L: all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ L = {}
  by simp

from A-prop [rule-format, of 0 p is $\emptyset$ sb $\mathcal{D} \mathcal{O}$] t
have sh-A: all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ A = {}
  by simp

from R-prop-acq [rule-format, of 0 p is $\emptyset$ sb $\mathcal{D} \mathcal{O}$] t
have acq-R: all-acquired (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ R = {}
  by simp

from R-prop-sh [rule-format, of 0 p is $\emptyset$ sb $\mathcal{D} \mathcal{O}$] t
have unsh-R: all-unshared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ R = {}
  by simp

from sharing-consis [of 0, simplified, OF t]
have sharing-consistent $\mathcal{S} \mathcal{O}$ sb.
from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent $\mathcal{S} \mathcal{O}$ (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb).

from share-commute [OF consis-sb sh-L sh-A acq-R unsh-R sh-R]
have share-eq:
  (share (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) ($\mathcal{S} \oplus_{W} R \ominus_{A} L$)) =
  (share (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\mathcal{S}$) $\oplus_{W} R \ominus_{A} L$.

let $?S' = (share (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\mathcal{S}$)

from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom $?S' \subseteq \mathcal{O}$.
from unshared-all-unshared [OF consis-sb] unshared-acquired-or-owned [OF consis-sb]
have unshared-acq-owned: dom $\mathcal{S} \subseteq \text{all-acquired} (\text{takeWhile} (\text{Not } \circ \text{ is-volatile-Write}_{sb}) \text{ sb}) \cup \mathcal{O}$
  by simp
have sep:
  \( \forall i < \text{length } ts \cdot \text{let } (-, -, r, sb', r, r) = ts!i \text{ in} \)
  all-acquired sb' \( \cap \) dom S \( - \) dom ?S' = \{\} \land
  all-unshared sb' \( \cap \) dom ?S' \( - \) dom S = \{\}

proof –

\[
\begin{align*}
\text{fix } i \text{ p is } \mathcal{O}, R, D, \emptyset, sb_i \\
\text{assume } i\text{-bound: } i < \text{length } ts \\
\text{assume } ts\text{-i: } ts ! i = (p_i, is_i, \emptyset, sb_i, D, \mathcal{O}, R) \\
\text{have all-acquired sb_i } \cap \text{ dom S } \text{ - dom } ?S' = \{\} \land
  \text{ all-unshared sb_i } \cap \text{ dom } ?S' \text{ - dom S} = \{\}
\end{align*}
\]

proof –

from ownership-distinct \(\text{[of 0 Suc i]}\) ts\(\text{-i } t \text{ i-bound} \)
have dist: \((\mathcal{O} \cup \text{all-acquired sb}) \cap (\mathcal{O}, \text{ all-acquired sb_i}) = \{\}
  \text{ by force}

from dist unshared-acq-owned all-acquired-takeWhile \(\text{[of (Not } \circ \text{ is-volatile-Write_{sb}}) \text{ sb]}\)
have all-acquired sb_i \( \cap \) dom S \( - \) dom ?S' = \{\}
  \text{ by blast}

moreover

from sharing-consis \(\text{[of Suc i]}\) ts\(\text{-i } t \text{ i-bound} \)
have sharing-consistent S \(\mathcal{O}_i\) sb_i
  \text{ by force}
from unshared-acquired-or-owned \(\text{[OF this]}\)
have all-unshared sb_i \( \subseteq \) all-acquired sb_i \( \cup \mathcal{O}_i\),
with dist fresh-owned
have all-unshared sb_i \( \cap \) dom ?S' - dom S = \{\}
  \text{ by blast}

ultimately show ?thesis by simp
  qed
  }
  thus ?thesis
  \text{ by } (fastforce simp add: Let-def)
  qed

from consis'.sharing-consis-preservation \(\text{[OF sep]}\)
have sharing-consis': sharing-consis (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}) S) ts.

from Cons.hyps \(\text{[OF dist' sharing-consis' L-prop' A-prop' R-prop' acq' R-prop' sh']}\)
have share-all-until-volatile-write ts ?S' \(\oplus_W\) R \(\ominus_A\) L =
  share-all-until-volatile-write ts (?S' \(\oplus_W\) R \(\ominus_A\) L).
then
have \(\text{share-all-until-volatile-write } ts\)

\(\forall S' \oplus_W R \ominus_A L = \text{share-all-until-volatile-write } ts\)

\((\text{share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} \ (S \oplus_W R \ominus_A L))\)

by (simp add: share-eq)

then

show ?case

by (simp add: t)

qed

lemma share-append-Ghost_{sb}:

\(\forall S. \\text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\} \implies (\text{share (sb @ [Ghost_{sb} A L R W])} S) = (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) } S) = (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} S)\)

apply (induct sb)

apply simp

subgoal for a sb S

apply (case-tac a)

apply auto

done

done

lemma share-append-Ghost_{sb}':

\(\forall S. \\text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\} \implies (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb @ [Ghost_{sb} A L R W])} S) = (\text{share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)} S)\)

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}: \(\text{outstanding-refs is-volatile-Write}_{sb} sb \neq \{\}\)

shows \(\forall S \ i \ [\text{ownership-distinct ts}; \ \text{sharing-consis } ts \ S]\)

\(i < \text{length ts} \ \text{ts}!i = (p', \text{is}', \theta', sb, D', O', R, D', O, R) \rightarrow\)

all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap L = \{\};

\(\forall j p \text{ is } O \ R \ D \ \forall sb. \ j < \text{length ts} \rightarrow i \neq j \rightarrow \text{ts}!j = (p, \text{is}, \theta, sb, D, O, R) \rightarrow\)

all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap \Lambda = \{\};

\(\forall j p \text{ is } O \ R \ D \ \forall sb. \ j < \text{length ts} \rightarrow i \neq j \rightarrow \text{ts}!j = (p, \text{is}, \theta, sb, D, O, R) \rightarrow\)

all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\};

\(\forall j p \text{ is } O \ R \ D \ \forall sb. \ j < \text{length ts} \rightarrow i \neq j \rightarrow \text{ts}!j = (p, \text{is}, \theta, sb, D, O, R) \rightarrow\)

all-unshared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\};

\(\forall j p \text{ is } O \ R \ D \ \forall sb. \ j < \text{length ts} \rightarrow i \neq j \rightarrow \text{ts}!j = (p, \text{is}, \theta, sb, D, O, R) \rightarrow\)

all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\}\

\implies \text{share-all-until-volatile-write } (\text{ts}[i := (p', \text{is}', \theta', sb @ [Ghost_{sb} A L R W], D', O')]) S = \text{share-all-until-volatile-write } ts S \oplus_W R \ominus_A L\)

323
proof (induct ts)
case Nil
thus case by simp
next
case (Cons t ts)
obtain pt is t R t D t acq t sb t where
t: t = (pt, is t, R t, D t, acq t, sb 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 L-prop: \( \forall j \) p is O R D \( \emptyset \) sb. \( j < \text{length} \ (t#ts) \) \( i \neq j \) \( \rightarrow \)
(t#ts)!j=(p, is, \emptyset, sb, D, O, R)
all-shared (takeWhile (Not \circ is-volatile-Write sb) sb) \cap L = \{\} by fact

have A-prop: \( \forall j \) p is O R D \( \emptyset \) sb. \( j < \text{length} \ (t#ts) \) \( i \neq j \) \( \rightarrow \)
(t#ts)!j=(p, is, \emptyset, sb, D, O, R)
all-shared (takeWhile (Not \circ is-volatile-Write sb) sb) \cap A = \{\} by fact

have R-prop-acq: \( \forall j \) p is O R D \( \emptyset \) sb. \( j < \text{length} \ (t#ts) \) \( i \neq j \) \( \rightarrow \)
(t#ts)!j=(p, is, \emptyset, sb, D, O, R)
all-acquired (takeWhile (Not \circ is-volatile-Write sb) sb) \cap R = \{\} by fact

have R-prop: \( \forall j \) p is O R D \( \emptyset \) sb. \( j < \text{length} \ (t#ts) \) \( i \neq j \) \( \rightarrow \)
(t#ts)!j=(p, is, \emptyset, sb, D, O, R)
all-unshared (takeWhile (Not \circ is-volatile-Write sb) sb) \cap 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.

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) sb t).

let \( S' \) = (share (takeWhile (Not \circ is-volatile-Write sb) sb t) S)
from freshly-shared-owned [OF consis-sb]

324
have fresh-owned: 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: dom \( S - \text{dom } ?S' \subseteq \text{all-acquired} \) (takeWhile \( \text{Not } \circ \text{is-volatile-Write}_{sb_t} \) sb) \( \cup \mathcal{O}_t \)
by simp

have sep:
\[ \forall i < \text{length } ts. \text{let } (\text{ts} ! i) = (p_i, \text{is}_i, \theta_i, \text{sb}_i, \text{D}_i, \text{O}_i, \text{R}_i) \]
\[ \text{have all-acquired } \text{sb}_i \cap \text{dom } S - \text{dom } ?S' = \{ \} \land \text{all-unshared } \text{sb}_i \cap \text{dom } ?S' - \text{dom } S = \{ \} \]
proof
- 
\{ 
fix i p_i \text{is}_i \text{O}_i \text{R}_i \text{D}_i \text{acq}_i \theta_i \text{sb}_i
assume i-bound: \( i < \text{length } ts \)
assume ts-i: ts ! i = (p_i, \text{is}_i, \theta_i, \text{sb}_i, \text{D}_i, \text{O}_i, \text{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 [of 0 Suc i] ts-i i-bound
have dist: (\( \text{O}_t \cup \text{all-acquired } \text{sb}_t \)) \( \cap (\text{O}_i \cup \text{all-acquired } \text{sb}_i) = \{ \} \)
by force

from dist unshared-acq-owned all-acquired-takeWhile [of (\( \text{Not } \circ \text{is-volatile-Write}_{sb_t} \)) sb]
have all-acquired \( \text{sb}_t \cap \text{dom } S - \text{dom } ?S' = \{ \} \)
by blast

moreover

from sharing-consis [of Suc i] ts-i i-bound
have sharing-consistent \( \text{O}_i \) \( \text{sb}_i \)
by force
from unshared-acquired-or-owned [OF this]
have all-unshared \( \text{sb}_i \subseteq \text{all-acquired } \text{sb}_i \cup \text{O}_i \).
with dist fresh-owned
have all-unshared \( \text{sb}_i \cap \text{dom } ?S' - \text{dom } S = \{ \} \)
by blast
ultimately show \( \text{thesis by simp} \)
qed

thus \( \text{thesis} \)
by (fastforce simp add: Let-def)
qed

from consis'.sharing-consis-preservation [OF sep]
have sharing-consis': sharing-consis [share (\( \text{takeWhile } \text{Not } \circ \text{is-volatile-Write}_{sb_t} \)) sb_t] \( \mathcal{S} \)
ts.
show ?case

proof (cases i)
  case 0
  with t Cons.prems have eqs: \( p_t = p \) \( \text{is}_t = \text{is} \) \( \mathcal{O}_t = \mathcal{O} \) \( \mathcal{R}_t = \mathcal{R} \) \( \mathcal{D}_t = \mathcal{D} \) \( \mathcal{D}_t = \mathcal{D} \)
  by auto

  from no-out-VWrite_{sb}
  have flush-all: takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \( \mathcal{D}_t \mapsto \emptyset \) sb = sb
  by (auto simp add: outstanding-refs-conv)

  from no-out-VWrite_{sb}
  have flush-all': takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \( \mathcal{D}_t \mapsto \emptyset \) \( \text{Ghost}_{sb} \ A L R W \) = \( \text{Ghost}_{sb} \ A L R W \)
  by (auto simp add: outstanding-refs-conv)

  have share-eq:
  (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \( \mathcal{D}_t \mapsto \emptyset \) \( \text{Ghost}_{sb} \ A L R W \))) \( \mathcal{S} \) =
  (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \( \mathcal{D}_t \mapsto \emptyset \) \( \text{Ghost}_{sb} \ A L R W \))) \( \mathcal{S} \) \( \oplus W R \ominus A L \)
  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':
  \( \forall i \ p \ \text{is} \ \mathcal{O} \ \mathcal{D} \ \emptyset \ sb. \)
  \( i < \text{length} \ \mathcal{D} \rightarrow \)
  \( \mathcal{D}_i = (p, \text{is}, \emptyset, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow \)
  \( \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \ \circ \ \text{is-volatile-Write}_{sb}) \ \mathcal{D}_t) \cap \mathcal{L} = \{} \)
  apply clarsimp
  subgoal for i1 p is \( \mathcal{O} \ \mathcal{D} \ \emptyset \ sb \)
  apply (drule-tac x=Suc i1 in spec)
  apply auto
  done
  done

  from A-prop 0 have A-prop':
  \( \forall i \ p \ \text{is} \ \mathcal{O} \ \mathcal{D} \ \emptyset \ sb. \)
  \( i < \text{length} \ \mathcal{D} \rightarrow \)
  \( \mathcal{D}_i = (p, \text{is}, \emptyset, \mathcal{D}, \mathcal{O}, \mathcal{R}) \rightarrow \)
  \( \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \ \circ \ \text{is-volatile-Write}_{sb}) \ \mathcal{D}_t) \cap \mathcal{A} = \{} \)
  apply clarsimp
  subgoal for i1 p is \( \mathcal{O} \ \mathcal{D} \ \emptyset \ sb \)
  apply (drule-tac x=Suc i1 in spec)
  apply auto
  done
  done

  from R-prop-acq 0 have R-prop-acq':
  \( \forall i \ p \ \text{is} \ \mathcal{O} \ \mathcal{D} \ \emptyset \ sb. \ i < \text{length} \ \mathcal{D} \rightarrow \)
  \( \text{all-acquired} \ (\text{takeWhile} \ (\text{Not} \ \circ \ \text{is-volatile-Write}_{sb}) \ \mathcal{D}_t) \cap \mathcal{R} = \{} \)
  apply clarsimp
  subgoal for i1 p is \( \mathcal{O} \ \mathcal{D} \ \emptyset \ sb \)

  326
apply (drule-tac x=Suc i1 in spec)
apply auto
done
done
from R-prop 0
have R-prop':
  \( \forall i \ p \ is \ ORD \ \theta \ sb. \ i < \text{length ts} \rightarrow ts!i=(p, is, \theta, sb, D, O, R) \rightarrow \)
  all-unshared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap R = {} \)
apply clarsimp
subgoal for i1 p is ORD \theta sb
apply (drule-tac x=Suc i1 in spec)
apply auto
done
done
from R-prop-sh 0 have R-prop-sh':
  \( \forall i \ p \ is \ ORD \ \theta \ sb. \ i < \text{length ts} \rightarrow ts!i=(p, is, \theta, sb, D, O, R) \rightarrow \)
  all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap R = {} \)
apply clarsimp
subgoal for i1 p is ORD \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']
have R-prop-sh' 

have share-all-until-volatile-write ts (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)
\( S \oplus_{W} R \oplus_{A} L = \)
share-all-until-volatile-write ts (share (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) 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 L-prop': \( \forall j \ p \ is \ ORD \ \theta \ sb. \ j < \text{length ts} \rightarrow k\neq j \rightarrow (ts)!j=(p, is, \theta, sb, D, O, R) \rightarrow \)
  all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap L = {} \ by \ force

from A-prop Suc
have A-prop': \( \forall j \ p \ is \ ORD \ \theta \ sb. \ j < \text{length ts} \rightarrow k\neq j \rightarrow (ts)!j=(p, is, \theta, sb, D, O, R) \rightarrow \)
  all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \cap A = {} \ by \ force
from R-prop-acq Suc have R-prop-acq':
\( \forall j \ p \ is \ ORD \ \theta \ sb. \ j < \text{length ts} \rightarrow k\neq j \rightarrow ts!j=(p, is, \theta, sb, D, O, R) \rightarrow \)
all-acquired \((\text{takeWhile } (\not\circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\}\) by force

from \textbf{R-prop Suc}
have \textbf{R-prop\textquotesingle}:
\[\forall j \text{ is } O \cap D \cap \not\circ \text{is-volatile-Write}_{sb} \text{. } j < \text{length ts} \rightarrow k \neq j \rightarrow ts[j] = (p, is, \theta, sb, D, O, R) \rightarrow\]
all-unshared \((\text{takeWhile } (\not\circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\}\) by force

from \textbf{R-prop-sh Suc} have \textbf{R-prop-sh\textquotesingle}:
\[\forall j \text{ is } O \cap D \cap \not\circ \text{is-volatile-Write}_{sb} \text{. } j < \text{length ts} \rightarrow k \neq j \rightarrow ts[j] = (p, is, \theta, sb, D, O, R) \rightarrow\]
all-shared \((\text{takeWhile } (\not\circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\}\) by force

from \textbf{Cons.prems Suc} obtain \textbf{k-bound: } k < \text{length ts} and \textbf{ts-k: } ts[k] = (p, is, \theta, sb, D, O, R)
by auto
from \textbf{Cons.hyps} \([\text{OF dist\textquotesingle} \text{sharing-consis\textquotesingle} \text{ k-bound ts-k L-prop\textquotesingle} \text{ A-prop\textquotesingle} \text{ R-prop-acq\textquotesingle} \text{ R-prop\textquotesingle} \text{ R-prop-sh\textquotesingle}])
show \textbf{?thesis}
by (clarsimp simp add: t Suc)
qed

lemma \textbf{share-domain-changes:}
\[\forall S \cap S\textquotesingle. \ a \in \text{all-shared sb } \cup \text{all-unshared sb } \Rightarrow \text{share sb } S\textquotesingle \ a = \text{share sb } S \ a\]
proof (induct sb)
case Nil thus \textbf{?case} by simp
next
case (\textbf{Cons x sb})
show \textbf{?case}
proof (cases x)
case (Write sb volatile a\textquotesingle sop v A L R W)
show \textbf{?thesis}
proof (cases volatile)
case True
note volatile=\this
from \textbf{Cons.prems obtain a-in: } a \in R \cup \text{all-shared sb } \cup L \cup \text{all-unshared sb}
by (clarsimp simp add: Write sb True)
show \textbf{?thesis}
proof (cases a \in R)
case True
from True have \((S\textquotesingle \oplus_W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a\)
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
from share-shared-eq \([\text{where } S\textquotesingle=S\textquotesingle \oplus_W R \ominus_A L \text{ and } S=S \oplus_W R \ominus_A L, \text{ OF this}]\)
have share sb \((S\textquotesingle \oplus_W R \ominus_A L) a = \text{share sb } (S \oplus_W R \ominus_A L) a\)
by auto

328
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: Ghost 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 \(a \in R \cup \text{all-shared sb} \cup L \cup \text{all-unshared sb}\)
    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 \[where S' = S' ⊕_W R ⊕_A L and S = S ⊕_W R ⊕_A L, OF this\]
  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: Ghost sb)
next
case False
note not-R = this
show ?thesis
proof (cases a ∈ 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 \((S' \oplus W R \ominus A L) a = \text{share sb } (S \oplus W R \ominus A L) a\)
by auto 
then show ?thesis
  by (clarsimp simp add: Ghost sb)
next 
case False 
with not-R a-in have \(a \in \text{all-shared sb } \cup \text{all-unshared sb}\)
by auto 
from Cons.hyps [OF this]
show ?thesis by (clarsimp simp add: Ghost sb)
qed 
qed 
qed

lemma share-domain-changesX:
\(\forall S \ S' X. \forall a \in X. \ S' a = S a\) 
\(\Rightarrow a \in \text{all-shared sb } \cup \text{all-unshared sb } \cup X \Rightarrow \text{share sb } S' a = \text{share sb } S a\)
proof (induct sb)
case Nil thus ?case by simp
next 
case (Cons x sb)
then have \(\forall a \in X. S' a = S a\)
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: \(a \in R \cup \text{all-shared sb } \cup L \cup \text{all-unshared sb } \cup X\)
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 \((S' \oplus W R \ominus A L) a = \text{share sb } (S \oplus W R \ominus A L) a\)
by auto 
then show ?thesis
  by (clarsimp simp add: Write sb volatile)
next 
case False 
note not-R = this
show \( ?\text{thesis} \)

proof (cases \( a \in L \))

\hspace{1em} 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 \( \text{share sb } (S' \oplus W R \ominus A L) a = \text{share sb } (S \oplus W R \ominus A L) a \)

by auto

then show \( ?\text{thesis} \)

by (clarsimp simp add: Write\(_{sb}\) volatile)

next

\hspace{1em} case False

from shared-eq have \( \text{shared-eq' } \forall a \in X. (S' \oplus W R \ominus A L) a = (S \oplus W R \ominus A L) a \)

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

from False not-R a-in have \( a \in \text{all-shared sb } \cup \text{all-unshared sb } \cup X \)

by auto

from Cons.hyps [OF shared-eq' this]

show \( ?\text{thesis} \) by (clarsimp simp add: Write\(_{sb}\) volatile)

qed

next

\hspace{1em} case False with Cons show \( ?\text{thesis} \) by (auto simp add: Write\(_{sb}\))

qed

next

\hspace{1em} case Read\(_{sb}\) with Cons show \( ?\text{thesis} \) by (auto)

next

\hspace{1em} case Prog\(_{sb}\) with Cons show \( ?\text{thesis} \) by (auto)

next

\hspace{1em} case (Ghost\(_{sb}\) A L R W)

from Cons.prems obtain \( a\text{-in: } a \in R \cup \text{all-shared sb } \cup L \cup \text{all-unshared sb } \cup X \)

by (clarsimp simp add: Ghost\(_{sb}\))

show \( ?\text{thesis} \)

proof (cases \( a \in R \))

\hspace{1em} 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 \( \text{share sb } (S' \oplus W R \ominus A L) a = \text{share sb } (S \oplus W R \ominus A L) a \)

by auto

then show \( ?\text{thesis} \)

by (clarsimp simp add: Ghost\(_{sb}\))

next

\hspace{1em} case False

note not-R = this

show \( ?\text{thesis} \)

proof (cases \( a \in L \))

\hspace{1em} 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$_{ab}$)
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 $a \in \text{all-shared sb} \cup \text{all-unshared sb} \cup X$
  by auto
from Cons.hyps [OF shared-eq' this]
show ?thesis by (clarsimp simp add: Ghost$_{ab}$)
qed
qed
qed

lemma share-unchanged:
\[ \forall S. a \notin \text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb} \Rightarrow \text{share sb } S a = S a \]
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write$_{ab}$ volatile a' sop 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': $a \notin \text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb}$
  by (clarsimp simp add: Write$_{ab}$ True)
from Cons.hyps [OF a']
have share sb $(S \oplus_W R \ominus_A L) a = (S \oplus_W R \ominus_A L) a$.
moreover
from a-R a-L a-A have $(S \oplus_W R \ominus_A L) a = S a$
  by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
ultimately
show ?thesis
  by (clarsimp simp add: Write$_{ab}$ 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\sb A L R W)
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: Ghost\sb)
from Cons.hyps [OF a\']
have share \sb (S \oplus W R \ominus A L) a = (S \oplus W R \ominus A L) a .
moreover
from a-R a-L a-A have (S \oplus W R \ominus 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 \cup L \cup A) \cap (all-shared \sb \cup all-unshared \sb \cup all-acquired \sb) = {}
shows (share \sb S \oplus W R \ominus A L) = share \sb (S \oplus W R \ominus A L)
proof —
from dist have shared-eq: \forall a \in all-acquired \sb. (S \oplus W R \ominus A L) a = S a
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
{
fix a
assume a-in: a \in all-shared \sb \cup all-unshared \sb \cup all-acquired \sb
from share-domain-changesX [OF shared-eq this]
have share \sb (S \oplus W R \ominus A L) a = share \sb S a.
also
from dist a-in have ... = (share \sb S \oplus W R \ominus A L) a
by (auto simp add: augment-shared-def restrict-shared-def split: option.splits)
finally have share \sb (S \oplus W R \ominus A L) a = (share \sb S \oplus W R \ominus A L) a.
}
moreover
{
fix a
assume a-notin: a \notin all-shared \sb \cup all-unshared \sb \cup all-acquired \sb
from share-unchanged [OF a-notin]
have share \sb (S \oplus W R \ominus A L) a = (S \oplus W R \ominus A L) a.
moreover
from share-unchanged [OF a-notin]
have share \sb S a = S a.
hence (share \sb 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)
ultimately have share \sb (S \oplus W R \ominus A L) a = (share \sb S \oplus W R \ominus A L) a
by simp
}
ultimately show ?thesis
apply —
apply (rule ext)
subgoal for x

qed
apply (case-tac x ∈ all-shared sb ∪ all-unshared sb ∪ all-acquired sb)
apply auto
done
done
qed

lemma share-append-commute:
\[ \forall ys S. (\text{all-shared } xs \cup \text{all-unshared } xs \cup \text{all-acquired } xs) \cap (\text{all-shared } ys \cup \text{all-unshared } ys \cup \text{all-acquired } ys) = \{\} \]

\[ \Rightarrow \text{share } xs (\text{share } ys S) = \text{share } ys (\text{share } xs S) \]

proof (induct xs)
  case Nil thus ?case by simp
next
case (Cons x xs)
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
  dist': (all-shared xs ∪ all-unshared xs ∪ all-acquired xs) ∩ (all-shared ys ∪ all-unshared ys ∪ all-acquired ys) = {\}
  apply (clarsimp simp add: Write sb True)
  apply blast
done
from Cons.prems have
dist: (R ∪ L ∪ A) ∩ (all-shared ys ∪ all-unshared ys ∪ all-acquired ys) = {\}
  apply (clarsimp simp add: Write sb True)
  apply blast
done
from share-augment-release-commute [OF dist] have (share ys S ⊕ W R ⊖ A L) = share ys (S ⊕ W R ⊖ A L).

  with Cons.hyps [OF dist']
  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 auto
next
case Prog sb with Cons show ?thesis by auto
next
case (Ghost sb A L R W)
from Cons.prems have
dist': (all-shared xs ∪ all-unshared xs ∪ all-acquired xs) ∩
(all-shared ys ∪ all-unshared ys ∪ all-acquired ys) = {}

apply (clarsimp simp add: Ghost
sb
)
apply blast
done
from Cons.prems have
dist: (R ∪ L ∪ A) ∩ (all-shared ys ∪ all-unshared ys ∪ all-acquired ys) = {}
apply (clarsimp simp add: Ghost
sb
)
apply blast
done
from share-augment-release-commute [OF dist]
have (share ys S ⊕ W R ⊖ A L) = share ys (S ⊕ W R ⊖ A L).

with Cons.hyps [OF dist]
show ?thesis
  by (clarsimp simp add: Ghost
sb
)
qed

done

lemma share-append-commute':
  assumes dist: (all-shared xs ∪ all-unshared xs ∪ all-acquired xs) ∩
  (all-shared ys ∪ all-unshared ys ∪ all-acquired ys) = {}
  shows share (ys@xs) S = share (xs@ys) S
proof
  from share-append-commute [OF dist] share-append [of xs ys] share-append [of ys xs]
  show ?thesis
    by simp
qed

lemma share-all-until-volatile-write-share-commute:
  shows \( \bigwedge S (sb': a \text{ memref list}) \) \([ownership-distinct ts; sharing-consis S ts;\]
  \( \forall i \) \( p \) \( is \) \( O \) \( R \) \( D \) \( \theta \) \( sb \) \( t \)
  \( i < \) \( length ts \)
  \( \rightarrow tsli=(p,is,\theta,\theta,\theta,\theta,\theta) \rightarrow \)
  (all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∪
  all-unshared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∪
  all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)) ∩
  (all-shared sb' ∪ all-unshared sb' ∪ all-acquired sb') = {}

\implies

share-all-until-volatile-write ts (share sb' S) =
share sb' (share-all-until-volatile-write ts S)

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 \( \theta t \) sb t where
    t: t=(p t,is t,\theta t,\theta t,\theta t,\theta t,\theta t)
  by (cases t)

  let ?take = (takeWhile (Not ◦ is-volatile-Write sb) sb)
  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: ∀i p is O R D ⊥ sb. i < length (t#ts)
→ (t#ts)!i=(p,is,⊥,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 sb' ∪ all-unshared sb' ∪ all-acquired sb') = {} by fact

from dist-prop [rule-format, of 0] t
have dist-t: (all-shared ?take ∪ all-unshared ?take ∪ all-acquired ?take) ∩
  (all-shared sb' ∪ all-unshared sb' ∪ all-acquired sb') = {} applies clarsimp

subgoal for i p is O R D ⊥ sb
apply (drule-tac x=Suc i in spec)
apply clarsimp
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 O₁ sb₁.

from sharing-consistent-takeWhile [OF this]
have consis-sb: sharing-consistent S O₁ ?take.

let ?S' = (share ?take S)

from freshly-shared-owned [OF consis-sb]
have fresh-owned: dom ?S' ⊆ O₁.
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 o is-volatile-Write sb) sb₁) ∪ O₁
by simp

have sep:
  \( \forall i < \text{length ts}. \text{let} \ (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) = \text{ts}!i \text{ in} \)
  all-acquired sb\( i' \cap \text{dom } S - \text{dom } ?S' = \{\} \wedge \)
  all-unshared sb\( i' \cap \text{dom } ?S' - \text{dom } S = \{\} \)
proof –
\{
  fix \( i \ p_i \) is\( i \) R\( i \) D\( i \) acq\( i \) \( \emptyset \) sb\( i \)
  assume i-bound: \( i < \text{length ts} \)
  assume ts-i: \( \text{ts} ! i = (p_i, \text{is}_i, \theta_i, \text{sb}_i, \text{D}_i, \text{O}_i, \text{R}_i) \)
  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: \( (\text{O}_t \cup \text{all-acquired sb}_t) \cap (\text{O}_i \cup \text{all-acquired sb}_i) = \{\} \)
    by force
  from dist unshared-acq-owned all-acquired-takeWhile [of (Not o is-volatile-Write sb) sb]
  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 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 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;
  \( \forall i \ p \ (p, is, \emptyset, sb, D, O, \mathcal{R}) \rightarrow \) ts!i = (p, is, \emptyset, sb, D, O, \mathcal{R}) →
    (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
    (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')) = {};
  \( \forall j \ p \ (p, is, \emptyset, sb, D, O, \mathcal{R}) \rightarrow \) ts!j = (p, is, \emptyset, sb, D, O, \mathcal{R}) →
    \( \forall i \ p \ (p, is, \emptyset, sb, D, O, \mathcal{R}) \rightarrow \)
  i \neq j → i < length ts →
\[(\text{all-shared } \text{sb} \cup \text{all-unshared } \text{sb} \cup \text{all-acquired } \text{sb}) \cap (R \cup L \cup A) = \{\}\]

\[\Rightarrow\]

share-all-until-volatile-write \((ts[i := (p',is',\theta',\text{sb}',D',O',R')])(S \oplus_W R \ominus_A L) =\]

share \((\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}')) \) \((\text{share-all-until-volatile-write } \text{ts} S \oplus_W R \ominus_A L)\)

**proof** (induct ts)

- case Nil
  - thus \(\text{case by simp}\)

- next
  - case \((\text{Cons } t \text{ ts})\)
    - obtain \(p_t, is_t, O_t, D_t, \theta_t, \text{sb}_t\) where
      \(t = (p_t, is_t, \theta_t, \text{sb}_t, D_t, O_t, R_t)\)
    - by \((\text{cases } t)\)

    - let \(\text{?take} = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}_t)\)
    - let \(\text{?take-sb}' = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb})\)
    - let \(\text{?drop} = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}_t)\)

    - have \(\text{dist: ownership-distinct } (t#\text{ts}) \text{ by fact}\)
    - then interpret \(\text{ownership-distinct } t#\text{ts} \text{ .}\)
    - have \(\text{consis: sharing-consis } S (t#\text{ts}) \text{ by fact}\)
    - then interpret \(\text{sharing-consis } S t#\text{ts} \text{ .}\)

    - have \(\text{dist-prop: } \forall i \text{ p is } O R D \emptyset \text{ sb}. i < \text{length } (t#\text{ts}) \rightarrow (t#\text{ts})!i = (p, is, \emptyset, D, O, R) \rightarrow \)
      \((\text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb})) \cup \text{all-unshared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}) \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}) \cap (\text{all-shared } \text{?take-sb'} \cup \text{all-unshared } \text{?take-sb'} \cup \text{all-acquired } \text{?take-sb'}) = \{\}\) by fact

    - from \(\text{dist-prop } [\text{rule-format, of } 0] t\)
    - have \(\text{dist-t: } (\text{all-shared } \text{?take} \cup \text{all-unshared } \text{?take} \cup \text{all-acquired } \text{?take}) \cap (\text{all-shared } \text{?take-sb'} \cup \text{all-unshared } \text{?take-sb'} \cup \text{all-acquired } \text{?take-sb'}) = \{\}\)
      - apply clarsimp
      - done

    - from \(\text{dist-prop have}\)
      \(\text{dist-prop': } \forall i \text{ p is } O R D \emptyset \text{ sb}. i < \text{length } \text{ts} \rightarrow \text{tsli} = (p, is, \emptyset, \text{sb}, D, O, R) \rightarrow (\text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb})) \cup \text{all-unshared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}) \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}) \cap (\text{all-shared } \text{?take-sb'} \cup \text{all-unshared } \text{?take-sb'} \cup \text{all-acquired } \text{?take-sb'}) = \{\}\)
      - apply (clarsimp)
      - subgoal for \(i \text{ p is } O R D \emptyset \text{ sb}\)
      - apply (drule-tac \(= \text{Suc } i \text{ in } \text{spec}\))
      - apply clarsimp
      - done

    - done

    - have \(\text{dist-prop-R-L-A: } \forall j \text{ p is } O R D \emptyset \text{ sb}. j < \text{length } (t#\text{ts}) \rightarrow i \neq j \rightarrow (t#\text{ts})!j = (p, is, \emptyset, \text{sb}, D, O, R) \rightarrow (\text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb}) \cap \)
\[(R \cup L \cup A) = \{\} \text{ by fact}\]

**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** aargh: (Not \circ is-volatile-Write sb) = (\lambda a. \neg is-volatile-Write sb 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 \ominus A L

**from** dist-prop-R-L-A 0 **have**
dist-prop-R-L-A': \forall i p is O R D \emptyset sb. i < length ts
\[\rightarrow t s l i = (p, is, \emptyset, sb, D, O, R) \rightarrow\]
(all-shared sb \cup all-unshared sb \cup all-acquired sb) \cap
(R \cup L \cup A) = \{\}
**apply** (clarsimp)
**subgoal for** i1 p is O R D \emptyset sb
**apply** (drule-tac x=Suc i1 in spec)
**apply** clarsimp
**done**
then
**have** dist-prop-R-L-A'': \forall i p is O R D \emptyset sb. i < length ts
\[\rightarrow t s l i = (p, is, \emptyset, sb, D, O, R) \rightarrow\]
(all-shared (takeWhile (Not \circ is-volatile-Write sb) sb) \cup all-unshared (takeWhile (Not \circ is-volatile-Write sb) sb) \cup
all-acquired (takeWhile (Not \circ is-volatile-Write sb) sb)) \cap
(R \cup L \cup A) = \{\}
**apply** (clarsimp)
**subgoal for** i p is O R D \emptyset sb
apply (cut-tac sb=sb in all-shared-takeWhile-subset [where P=Not ◦ is-volatile-Write\sb])
apply (cut-tac sb=sb in all-unshared-takeWhile-subset [where P=Not ◦ is-volatile-Write\sb])
apply (cut-tac sb=sb in all-acquired-takeWhile-subset [where P=Not ◦ is-volatile-Write\sb])
apply fastforce
done
done

have sep: ∀i<length ts.
  let (\_\_, \_\_, \_\_, sb\_\_, \_\_, \_\_, \_\_) = ts ! i
  in ∀a∈all-acquired sb. ?S' a = S a
proof –
  \{
    fix i p_i is_i R_i D_i acqu_i \theta_i sb_i a
    assume i-bound: i < length ts
    assume ts-i: ts ! i = (p_i, is_i, \theta_i, sb_i, D_i, O_i, R_i)
    assume a-in: a ∈ all-acquired sb_i
    have ?S' a = S a
    proof –
      from dist-prop-R-L-A' [rule-format, OF i-bound ts-i] a-in
      show ?thesis
        by (auto simp add: augment-shared-def restrict-shared-def split: option.split)
    qed
  }
thus ?thesis by auto
qed
from consis'.sharing-consis-shared-exchange [OF sep]
have sharing-consis': sharing-consis ?S' ts.

from share-all-until-volatile-write-share-commute [of ts (S ⊕ W R ⊕ A L) (takeWhile (Not ◦ is-volatile-Write\sb) sb')], OF dist' sharing-consis' dist-prop']

have share-all-until-volatile-write ts (share ?take-sb' ?S') =
  share ?take-sb' (share-all-until-volatile-write ts ?S') .

moreover

from dist-prop-R-L-A''
have (share-all-until-volatile-write ts (S ⊕ W R ⊕ A L)) =
  (share-all-until-volatile-write ts S ⊕ W R ⊕ A L)

apply –
  apply (rule share-all-until-volatile-write-commute [OF dist' consis', of L A R W,symmetric])
  apply (clarsimp blast)+
done
ultimately
show ?thesis

341
using takeWhile-empty
by (clarsimp simp add: t 0 aargh eqs)

next
  case (Suc k)
  from Cons.prems Suc obtain k-bound: k < length ts and ts-k: ts!k = (p, is, θ, sb, D, O, R)
  by auto

  let ?S' = (share (takeWhile (Not ◦ is-volatile-Write sb) sb t) S)
  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

  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 (-, -, -, 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 O_i R_i D_i acqi q_i sb_i
    assume i-bound: i < length ts
    assume ts-i: ts ! i = (p_i, is_i, q_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_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_i]
    have all-acquired sb_i ∩ dom S − dom ?S' = {}
    by blast

    moreover
    from sharing-consis [of Suc i] ts-i t i-bound

342
have sharing-consistent \( S \subseteq O_i \) \( sb_i \)

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 \( \text{thesis by simp} \)

qed

}  

thus \( \text{thesis} \)

by (fastforce simp add: Let-def)

qed

from consis'.sharing-consis-preservation [OF sep]

have sharing-consis' \( : \text{sharing-consis } ?S' \) ts.

from dist-prop-R-L-A [rule-format, of 0] Suc t

have dist-t-R-L-A: (all-shared \( sb_t \cup \) all-unshared \( sb_t \cup \) all-acquired \( sb_t \) ) \( \cap \)

(\( R \cup L \cup A \) ) = {}  

apply clarsimp done

from dist-t-R-L-A

have \( (R \cup L \cup A) \cap (\text{all-shared } ?\text{take} \cup \text{all-unshared } ?\text{take} \cup \text{all-acquired } ?\text{take}) = {} \)

using all-unshared-append [of \( ?\text{take} ?\text{drop} \) ]

all-acquired-append [of \( ?\text{take} ?\text{drop} \) ]

by auto

from share-augment-release-commute [OF this]

have share \( ?\text{take } S \oplus_{W} R \ominus_{A} L = \text{share } ?\text{take } (S \oplus_{W} R \ominus_{A} L) . \)

moreover

from dist-prop-R-L-A Suc

have \( \forall j \text{ p is } O \( \cap \) R \( \cap \) D \( \neq \) sb, \( j < \text{length } (ts) \) \( \rightarrow \) k \( \neq \) j

\( \rightarrow (ts)!j=(p, is, \theta, sb, D, O, R) \rightarrow \)

(all-shared sb \( \cup \) all-unshared sb \( \cup \) all-acquired sb) \( \cap \)

(R \( \cup \) L \( \cup \) A ) = {} 

apply (clarsimp)

subgoal for \( j \text{ p is } O \( \cap \) R \( \cap \) D \( \neq \) 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 \( \text{thesis} \)

by (clarsimp simp add: t Suc)

qed
lemma share-all-until-volatile-write-Ghost\_sb-commute:
shows \( \forall S i. \) \{ownership-distinct ts; sharing-consis \( S \) ts; \( i < \) length ts; 
\( \text{ts}!i = (p, i, s, \text{Ghost}_{sb} A L R W \# sb, D, O, R) \) \} 

\( \implies \) share-all-until-volatile-write \((\text{ts}[i := (p', i', s', \text{sb})]) S \) \( S' \cup W R \ominus A \) = \{\} \]

proof (induct ts)
  case Nil
  thus \( ? \) case by simp
next
  case (Cons \( t \) \( \text{ts} \))
  obtain \( p_t \) \( i_t \) \( O_t \) \( D_t \) \( v_t \) \( \text{sb}_t \) where 
  \( t : t = (p_t, i_t, v_t, \text{sb}_t, D_t, O_t, R_t) \) 
  by (cases \( t \))
  have \( \text{dist} : \) ownership-distinct \((t \# \text{ts})\) by fact
  then interpret ownership-distinct \((t \# \text{ts})\).
  have \( \text{consis} : \) sharing-consis \( S (t \# \text{ts})\) by fact
  then interpret sharing-consis \( S t \# \text{ts} \).
  have \( \text{dist-prop} : \) \( \forall p j. \) \( j < \) length \((t \# \text{ts})\) \( i \neq j \) \( \text{ts}[j] = (p, i, s, \text{sb}) \) \( \# sb, D, O, R) \) \( \implies \) 
  (all-shared \((\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \)) \( \cup \) all-unshared \((\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \)) \( \) \( \cap \)
  
  \( (R \cup L \cup A) = \{\} \) by fact

from ownership-distinct-tl [OF \( \text{dist} \) ]
have \( \text{dist}' : \) ownership-distinct \( \text{ts} \).

from sharing-consis-tl [OF \( \text{consis} \) ]
have \( \text{consis}' : \) sharing-consis \( S \) \( \text{ts} \).
then
interpret \( \text{consis}' : \) sharing-consis \( S \) \( \text{ts} \).

from sharing-consis [of 0, simplified, OF \( t \) ]
have \( \text{sharing-consistent} \( S O_t \) \( \text{sb}_t \).

from sharing-consistent-takeWhile [OF \( \text{this} \) ]
have \( \text{consis-sb} : \) sharing-consistent \( S O_t \) \((\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \).

let \( ?S' = (\text{share} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) S) \)

from freshly-shared-owned [OF \( \text{consis-sb} \) ]
have \( \text{fresh-owned} : \) 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 $S - dom ?S' \subseteq$ all-acquired (takeWhile (Not $(\circ)$ is-volatile-Write $sb_t$) $sb_t$) $\cup O_t$

by simp

have sep:
$\forall i < \text{length } ts \cdot \text{let } (p_i, is_i, D_i, O_i) = ts!i \text{ in}$
all-acquired $sb'_i \cap dom S - dom ?S' = \{} \wedge$
all-unshared $sb'_i \cap dom ?S' - dom S = \{}$

proof –
{
fix $i$, $p_i$, $is_i$, $D_i$, $O_i$, $sb_i$
assume i-bound: $i < \text{length } ts$
assume ts-i: $ts!i = (p_i, is_i, D_i, O_i, R_i)$
have all-acquired $sb_i \cap dom S - dom ?S' = \{} \wedge$
all-unshared $sb_i \cap dom ?S' - dom S = \{}$

proof –
from ownership-distinct [of 0 Suc i] ts-i t i-bound
have 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 $(\circ)$ is-volatile-Write $sb_t$) $sb_t$]

have all-acquired $sb_i \cap dom S - dom ?S' = \{}$
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 all-unshared $sb_i \subseteq \text{all-acquired } sb_i \cup O_i$,
with dist fresh-owned
have all-unshared $sb_i \cap dom ?S' - dom S = \{}$
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 (share (takeWhile (Not $(\circ)$ is-volatile-Write $sb_t$) $sb_t$) $S$) $ts$.

show ?case
proof (cases i)

  case 0

    with t Cons.prems have eqs: p t = p is t = is t = is t = is t = is t = is t = is t = is t = is ∪ D t = 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, ∅, Ghost sb A L R W # sb D)
  by auto

  from dist-prop Suc have dist-prop': ∀ j p isORD θ sb. j < length ts → k ≠ j → ts[k = (p, is, ∅, Ghost sb A L R W # sb D)
  by auto

  show ?thesis
  by (clarsimp)

  from Cons.hyps [OF dist' sharing-consis' k-bound ts-k dist-prop'] have share-all-until-volatile-write (ts[k := (p', is', ∅, sb, D', O', R')])
  (share (takeWhile (Not o is-volatile-Write sb) sb t) S ⊕ W R ⊕ A L) =
  share-all-until-volatile-write ts
  (share (takeWhile (Not o is-volatile-Write sb) sb t) S).

  moreover
  from dist-prop [rule-format, of 0 p t is t = ∅ sb t D t O t R t] t Suc have (R ∪ L ∪ A) ∩ (all-shared (takeWhile (Not o is-volatile-Write sb) sb t) ∪ all-unshared (takeWhile (Not o is-volatile-Write sb) sb t) ∪ all-acquired (takeWhile (Not o is-volatile-Write sb) sb t)) = {}
    apply clarsimp
    apply blast
    done
  from share-augment-release-commute [OF this] have share (takeWhile (Not o is-volatile-Write sb) sb t) S ⊕ W R ⊕ A L =
  share (takeWhile (Not o is-volatile-Write sb) sb t) (S ⊕ W R ⊕ A L).

  ultimately
  show ?thesis
  by (clarsimp simp add: Suc t)
qed
lemma share-all-until-volatile-write-update-sb:
assumes congr: \( \forall S. \) share (takeWhile (Not \circ is-volatile-Write_{sb}) sb') S = share (takeWhile (Not \circ is-volatile-Write_{sb}) sb) S
shows \( \forall S i. [ i < \text{length ts}; \text{ts![i] = (p, is, } \theta, \text{sb, D, } O, R] ] \) \( \Rightarrow \)
share-all-until-volatile-write ts S = share-all-until-volatile-write (ts[i := (p', is', } \theta', \text{sb', D', } O', R')]) S
proof (induct ts)
case Nil
thus ?case by simp
next
case (Cons t ts)
obtain \( p_t is_t O_t D_t } \theta_t sb_t \text{ where} \)
t: t=(p_t,is_t,\theta_t,\text{sb}_t,\text{D}_t,\text{O}_t,\text{R}_t)
by (cases t)
show ?case
proof (cases i)
case 0
with t Cons.prems have eqs: \( p_t=p \text{ is}_t=is O_t=O \text{ \theta}_t=\theta \text{ sb}_t=\text{sb} \text{ D}_t=D \)
by auto
show ?thesis
by (clarsimp simp add: 0 t eqs congr)
next
case (Suc k)
from Cons.prems Suc obtain k-bound: k < \text{length ts} and ts-k: ts![k] = (p, is, } \theta, \text{sb, D, O, R})
by auto
from Cons.hyps [OF k-bound ts-k ]
show ?thesis
by (clarsimp simp add: t Suc)
qed
qed

lemma share-all-until-volatile-write-append-Ghost_{sb}:
assumes out-VWrite_{ab}: outstanding-refs is-volatile-Write_{sb} sb \neq \{ \}
assumes i-bound: i < \text{length ts}
assumes ts-i: ts![i] = (p, is, } \theta, \text{sb, D, O, R})
shows share-all-until-volatile-write ts S =
share-all-until-volatile-write (ts[i := (p', is', } \theta', \text{sb', D', } O', R')]) S
proof
from out-VWrite_{sb}
have \( \forall S. \) share (takeWhile (Not \circ is-volatile-Write_{sb}) (sb @ [Ghost_{sb} A L R W])) S =
share (takeWhile (Not \circ is-volatile-Write_{sb}) 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:
\( \forall S. \ (\text{acquired pending-write} \ (\text{takeWhile} \ (\text{Not} \ \text{is-volatile-Write}_{sb}) \ (sb \ @ \ [\text{Prog}_{sb} p_1 p_2 \ \text{mis}])) \ S) = \ \\
(\text{acquired pending-write} \ (\text{takeWhile} \ (\text{Not} \ \text{is-volatile-Write}_{sb}) \ sb) \ S) \ \\
\) by (induct sb) (auto split: memref.splits)

lemma outstanding-refs-non-empty-dropWhile:
outstanding-refs P xs \neq \{} \implies \text{outstanding-refs} P \ (\text{dropWhile} \ (\text{Not} \ \circ \ 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 θ sb \( D \ O \ \mathcal{R} \) p' is' )
then obtain
  i-bound: \( i < \text{length} \ ts \) and
  ts-i: \( ts!i = (p,\text{is},\theta,\text{sb},D,O,\mathcal{R}) \) and
  prog-step: \( \theta \vdash p \rightarrow p' (p',\text{is}') \) and
  ts': \( ts'=ts[i:=(p',\text{is}@\text{is}',\theta,\text{record} \ p \ p' \ \text{is}' \ \text{sb},D,O,\mathcal{R})] \) and
  \( S' = S \) and
  m': \( m'=m \) 
by auto

from i-bound have i-bound': \( i + \text{length} \ xs < \text{length} \ (xs@ts) \) 
  by auto

from ts-i i-bound have ts-i': \( (xs@ts)[i+\text{length} \ xs] = (p,\text{is},\theta,\text{sb},D,O,\mathcal{R}) \)
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' sb' D' O' R')
then obtain
  i-bound: i < length ts and
  ts-i: ts!i = (p, is, sb, D, O, R)
  memop-step: (is, sb, m, D, O, R, S) →_m (is', sb', m', D', O', R', S')
  ts': ts' = ts[i:=i=p, is, 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, sb, D, O, R)
  by (auto simp add: nth-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, sb, D, O, R)
  sb-step: (m, sb, O, R, S) →_sb (m', sb', O', R', S')
  ts': ts' = ts[i:=i=p, is, 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, 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)

qed

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
by (auto simp add: nth-append)
else weak-sharing-consistent \( \mathcal{O} \) \( rs \)
\[ | \text{Ghost}_{sb} A L R W \Rightarrow L \subseteq A \land A \cap R = \{ \} \land R \subseteq \mathcal{O} \land A - R \) \( rs \)
\[ | \Rightarrow \text{weak-sharing-consistent } \mathcal{O} \) \( rs \)

**Lemma** sharing-consistent-weak-sharing-consistent:
\[ \bigwedge \mathcal{S} \mathcal{O}. \text{sharing-consistent } \mathcal{S} \mathcal{O} \) \( sb \equiv \text{weak-sharing-consistent } \mathcal{O} \) \( sb \)
**apply** (induct \( sb \))
**apply** (auto split: memref.splits)
**done**

**Lemma** weak-sharing-consistent-append:
\[ \bigwedge \mathcal{O}. \text{weak-sharing-consistent } \mathcal{O} \) \( (xs \oplus ys) = \)
\[ \text{weak-sharing-consistent } \mathcal{O} \) \( xs \land \text{weak-sharing-consistent } \) \( \) \( \) \( \text{acquired True } \) \( xs \) \( \mathcal{O} \) \( ys \)
**apply** (induct \( xs \))
**apply** (auto split: memref.splits)
**done**

**Lemma** read-only-share-unowned:
\[ \bigwedge \mathcal{O} \mathcal{S}. \]
\[ \) \( \text{weak-sharing-consistent } \mathcal{O} \) \( sb; a \notin \mathcal{O} \cup \text{all-acquired } \) \( sb; a \in \text{read-only } \) \( \text{(share } \) \( sb \) \( \mathcal{S} \)]
\[ \Rightarrow a \in \text{read-only } \mathcal{S} \]
**proof** (induct \( sb \))
\[ \) \( \text{case Nil thus } \) ?case by simp
**next**
\[ \) \( \text{case } (\text{Cons } x \) \( sb \))
\[ \) \( \text{show } ?\text{case}
\[ \) \( \text{proof } (\text{cases } x)
\[ \) \( \text{case } (\text{Write}_{sb} \) \( \text{volatile } a' \) \( \text{sop v A L R W} \)
\[ \) \( \text{show } ?\text{thesis}
\[ \) \( \text{proof } (\text{cases } \text{volatile})
\[ \) \( \text{case False}
\[ \) \( \text{with Cons } \text{Write}_{sb} \) \( \text{show } ?\text{thesis by auto}
\[ \) \( \text{next}
\[ \) \( \text{case True}
\[ \) \( \text{from Cons.hyps } [\text{where } \mathcal{S}=\langle S \oplus_{W} R \ominus_{A} L \rangle \text{ and } \mathcal{O}=\langle \mathcal{O} \cup A - R \rangle ] \) \( \text{Cons.prems}
\[ \) \( \text{show } ?\text{thesis}
\[ \) \( \text{by } (\text{auto simp add: Write}_{sb} \text{ True in-read-only-restrict-conv in-read-only-augment-conv split: if-split-asm})
\[ \) \( \text{qed}
**next**
\[ \) \( \text{case } \text{Read}_{sb} \) \( \text{with Cons } \text{show } ?\text{thesis by auto}
\[ \) \( \text{next}
\[ \) \( \text{case } \text{Prog}_{sb} \) \( \text{with Cons } \text{show } ?\text{thesis by auto}
\[ \) \( \text{next}
\[ \) \( \text{case } (\text{Ghost}_{sb} A L R W)
\[ \) \( \text{with Cons.hyps } [\text{where } \mathcal{S}=\langle S \oplus_{W} R \ominus_{A} L \rangle \text{ and } \mathcal{O}=\langle \mathcal{O} \cup A - R \rangle ] \) \( \text{Cons.prems show}
\[ \) \( ?\text{thesis}
\[ \) \( \text{by } (\text{auto simp add: in-read-only-restrict-conv in-read-only-augment-conv split: if-split-asm})
\[ \) \( \text{qed}
\]
lemma share-read-only-mono-in:
assumes a-in: a ∈ read-only (share sb S)
assumes ss: read-only S ⊆ read-only S′
shows a ∈ read-only (share sb S′)
using share-read-only-mono [OF ss] a-in
by auto

lemma read-only-unacquired-share:
\[ \forall S. [O \cap read-only S = \{\}; weak-sharing-consistent O sb; a \in read-only S; 
a \notin all-acquired sb ] \]
\[ \Rightarrow a \in read-only (share sb S) \]
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-A: a \notin A and a-unacq: a \notin all-acquired sb and
  owns-ro: O \cap read-only S = \{\} and
  L-A: L ⊆ A and A-R: A \cap R = \{\} and R-owns: R ⊆ O and
  consis′: weak-sharing-consistent (O \cup A - R) sb
  by (clarsimp simp add: Write sb True)
  from owns-ro A-R owns 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 a-ro a-a owns-ro R-owns L-A have a-ro′: a ∈ 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′ 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 \not\in A and a-unacq: a \not\in all-acquired sb 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 a-ro a-A owns-ro R-owns L-A have a-ro': a ∈ 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' a-unacq]
show ?thesis
by (clarsimp simp add: Ghost sb)
qed
qed

lemma read-only-share-unacquired: \bigwedge \O \S. \O \cap read-only S = {} \implies
weak-sharing-consistent \O sb \implies a \in read-only (share sb S) \implies a \not\in 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 False
with Cons Write sb show ?thesis by auto
next
case True
note volatile=this
from Cons.prems
obtain a-ro: a ∈ 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

352
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

definition 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 \(S\))]

\(\equiv\) 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 \(\in\) read-only (share sb (\(S \oplus W R \ominus A L\))) and
owns-ro: $\mathcal{O} \cap \text{read-only } S = \{\}$ and
L-A: $L \subseteq A$ and A-R: $A \cap R = \{\}$ and R-owns: $R \subseteq \mathcal{O}$ and
consis': weak-sharing-consistent $(\mathcal{O} \cup A - R)$ sb
by (clarsimp simp add: Write sb True)

from owns-ro A-R R-owns have owns-ro': $(\mathcal{O} \cup A - R) \cap \text{read-only } (S \ominus W R \ominus L) = \{\}$
by (auto simp add: in-read-only-convs)

from Cons.hyps [OF owns-ro' consis' a-in]
have hyp: $a \in \text{read-only } (\text{share sb } \text{Map.empty}) \lor a \in \text{read-only } (S \ominus W R \ominus L) \land a \notin \text{all-acquired sb}$

have $a \in \text{read-only } (\text{share sb } (\text{Map.empty} \ominus W R \ominus L)) \lor (a \in \text{read-only } S \land a \notin A \land a \notin \text{all-acquired sb})$
proof -
{
assume a-emp: $a \in \text{read-only } (\text{share sb } \text{Map.empty})$
have read-only Map.empty $\subseteq$ read-only $(\text{Map.empty} \ominus W R \ominus L)$
by (auto simp add: in-read-only-convs)

from share-read-only-mono-in [OF a-emp this]
have a $\in$ read-only (share sb (Map.empty $\ominus W R \ominus L)$).
}
moreover
{
assume a-ro: $a \in$ read-only $(S \ominus W R \ominus L)$ and a-unacq: $a \notin$ all-acquired sb
have ?thesis
proof (cases $a \in$ read-only S)
  case True
  with a-ro obtain $a \notin A$
  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: $a \in$ read-only $(\text{Map.empty} \ominus W R \ominus L)$
  by (auto simp add: in-read-only-convs split: if-split-asm)

  have read-only $(\text{Map.empty} \ominus W R \ominus L) \subseteq$ read-only $(S \ominus W R \ominus L)$
  by (auto simp add: in-read-only-convs)
  with owns-ro'
  have owns-ro-empty: $(\mathcal{O} \cup A - R) \cap \text{read-only } (\text{Map.empty} \ominus W R \ominus L) = \{\}$
  by blast

from read-only-unacquired-share [OF owns-ro-empty consis' a-ro-empty a-unacq]
have $a \in$ read-only (share sb (Map.empty $\ominus W R \ominus L)$).
thus ?thesis
by simp

354
qed

moreover note hyp
ultimately show \( ?\text{thesis} \) by blast

due

then show \( ?\text{thesis} \)
by (clarsimp simp add: Write_{sb} True)

next

case False with Cons show \( ?\text{thesis} \)
by (auto simp add: Write_{sb})

qed simp add: Write

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 a-in: \( a \in \text{read-only (share sb (S } \ominus W R \ominus A L)) \) 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': \( (O \cup A - R) \cap \text{read-only (S } \ominus W R \ominus A L) \)
= \{\}

by (auto simp add: in-read-only-convs)

from Cons.hyps [OF owns-ro' consis' a-in]

have hyp: \( a \in \text{read-only (share sb Map.empty)} \) ∨ \( a \in \text{read-only (S } \ominus W R \ominus A L) \) \(\land a \notin \text{all-acquired sb} \)


have a \( \in \) read-only (share sb (Map.empty \( \ominus 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: \( a \in \text{read-only (share sb Map.empty)} \)

have read-only Map.empty \( \subseteq \) read-only (Map.empty \( \ominus 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 \( \in \) read-only (share sb (Map.empty \( \ominus W R \ominus A L)))


} moreover

{
assume a-ro: \( a \in \text{read-only (S } \ominus W R \ominus A L) \) and a-unacq: \( a \notin \text{all-acquired sb} \)

have ?thesis

proof (cases a \( \in \) read-only S)

case True

355
with a-ro obtain a \notin 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: \( a \in \text{read-only} \) \( (\text{Map}.\text{empty} \oplus_{\text{W}} \text{R} \ominus_{\text{A}} \text{L}) \)
  by (auto simp add: in-read-only-convs split: if-split-asm)

have read-only \( (\text{Map}.\text{empty} \oplus_{\text{W}} \text{R} \ominus_{\text{A}} \text{L}) \subseteq \text{read-only} \) \( (\text{S} \oplus_{\text{W}} \text{R} \ominus_{\text{A}} \text{L}) \)
  by (auto simp add: in-read-only-convs)
with owns-ro'
have owns-ro-empty: \((\mathcal{O} \cup \text{A} \setminus \text{R}) \cap \text{read-only} \) \( (\text{Map}.\text{empty} \oplus_{\text{W}} \text{R} \ominus_{\text{A}} \text{L}) = \{\} \)
  by blast

from read-only-unacquired-share \[ \text{OF owns-ro-empty consis'} a-ro-empty a-unacq \]
have a \( \in \text{read-only} \) \( (\text{share sb} (\text{Map}.\text{empty} \oplus_{\text{W}} \text{R} \ominus_{\text{A}} \text{L})) \).
thus ?thesis
  by simp
qed

moreover note hyp
ultimately show ?thesis by blast
qed
then show ?thesis
  by (clarsimp simp add: Ghost\_sb)
qed
qed

lemma weak-sharing-consistent-preserves-distinct:
\( \forall \mathcal{O} \mathcal{S}. \text{weak-sharing-consistent} \mathcal{O} \text{ sb } \Longrightarrow \mathcal{O} \cap \text{read-only} \mathcal{S} = \{\} \Longrightarrow \)
acquired True sb \( \mathcal{O} \cap \text{read-only} \) \( (\text{share sb} \mathcal{S}) = \{\} \)
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: \( \mathcal{O} \cap \text{read-only} \mathcal{S} = \{\} \) and L-A: \( \text{L} \subseteq \text{A} \) and A-R: \( \text{A} \cap \text{R} = \{\} \) and
R-owns: \( \text{R} \subseteq \mathcal{O} \) and consis': weak-sharing-consistent \( (\mathcal{O} \cup \text{A} \setminus \text{R}) \) sb

356
by (clarsimp simp add: Write sb True)

from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (S \oplus_W R \ominus_A L)\) = {}
by (auto simp add: in-read-only-convs)
from Cons.hyps [OF 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: \(\mathcal{O} \cap \text{read-only} S = \{\}\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and
R-owns: \(R \subseteq \mathcal{O}\) and consis': weak-sharing-consistent \((\mathcal{O} \cup A - R)\) sb
by (clarsimp simp add: Ghost sb)

from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (S \oplus_W R \ominus_A L)\) = {}
by (auto simp add: in-read-only-convs)
from Cons.hyps [OF 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:
\(\forall i\ p\ is\ \mathcal{O} \ R \ D \ \emptyset\ sb.\)
\([i < \text{length ts};\ ts!i = (p,\text{is},\emptyset,\text{sb},D,\mathcal{O},R)\ ]\)
\(\implies\)
weak-sharing-consistent \(\mathcal{O}\) sb

sublocale sharing-consis \(\subseteq\) weak-sharing-consis

proof
fix i p is \(\mathcal{O} \ R \ D \ \emptyset\) sb
assume i-bound: \(i < \text{length ts}\)
assume ts-i: ts ! i = (p, is, \emptyset, sb, D, \mathcal{O},R)
from sharing-consistent-weak-sharing-consistent [OF sharing-consis [OF i-bound ts-i]]
show weak-sharing-consistent \(\mathcal{O}\) sb.
qed
lemma weak-sharing-consis-tl: weak-sharing-consis (t#ts) \implies weak-sharing-consis ts
apply (unfold weak-sharing-consis-def)
apply force
done

lemma read-only-share-all-until-volatile-write-unacquired:
\forall S. \{\text{ownership-distinct ts}; \text{read-only-unowned } S \text{ ts}; \text{weak-sharing-consis ts};
\forall i < \text{length ts}. (\text{let } (r, r', s_b, o, r) = \text{ts!i in}
    a \notin \text{all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{ab}) s_b));
    a \in \text{read-only } S]\}
\implies a \in \text{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 ORD \theta sb where
t: t = (p, is, \theta, 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 \circ \text{is-volatile-Write}_{ab}) = (\lambda a. \neg \text{is-volatile-Write}_{ab} a)
by (rule ext) auto

have a-ro: a \in \text{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 \circ \text{is-volatile-Write}_{ab}) s_b)
let ?drop-sb = (dropWhile (Not \circ \text{is-volatile-Write}_{ab}) s_b)

from weak-sharing-consis [of 0] t
have consis-sb: weak-sharing-consistent O s_b
    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 : O_j ∩ D_j ∩ φ_j sb_j
assume j-bound: j < length ts
assume j-th: ts!j = (p_j, φ_j sb_j, D_j, O_j, R_j)
show O_j ∩ read-only (share ?take-sb S) = {}
have a ∈ read-only (share (takeWhile (Not \circ is-volatile-Write_{sb}) sb) S).
from Cons.hyps [OF dist’ ro-unowned’ consis’ unacq-ts this]
show ?case
  by (simp add: t)
qed

lemma read-only-share-unowned-in:
[weak-sharing-consistent \O sb; a ∈ read-only (share sb S)]
⇒ a ∈ read-only S ∪ \O ∪ all-acquired sb
using read-only-share-unowned [of \O sb]
by auto

lemma read-only-shared-all-until-volatile-write-subset:
\\land S. [ownership-distinct ts;
  weak-sharing-consis ts] ⇒
read-only (share-all-until-volatile-write ts S) ⊆
read-only S ∪ (\ union ((\lambda (\cdot, \cdot, \cdot, \cdot, sb, \cdot, \cdot, \cdot, \O, \cdot)). \O ∪ all-acquired (takeWhile (Not \circ is-volatile-Write_{sb}) 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, \emptyset, 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 consis: weak-sharing-consis (t#ts) by fact
then interpret weak-sharing-consis t#ts .

have aargh: (Not \circ is-volatile-Write_{sb}) = (\lambda a. ¬ is-volatile-Write_{sb} a)
  by (rule ext) auto

note consis’ = weak-sharing-consis-tl [OF consis]

let ?take-sb = (takeWhile (Not \circ is-volatile-Write_{sb}) sb)
let ?drop-sb = (dropWhile (Not \circ 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 \in \text{read-only}  
      \begin{align*}
      & (\text{share-all-until-volatile-write ts} \quad (\text{share ?take-sb } S)) \quad \text{and} \\
      & a-\text{notin-shared: } a \notin \text{read-only } S \quad \text{and} \\
      & a-\text{notin-rest: } a \notin (\bigcup (\lambda (-, -, -, sb, -, O, R). \quad O \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab} ) \text{ sb}) \quad \set ts))  
   \end{align*}

  \begin{align*}
  & \have a \in O \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab} ) \text{ sb}) \\
  \begin{proof} 
  \begin{itemize}
  \item \from \text{Cons.hyps } [\OF \dist' \consis', of \(\text{share ?take-sb } S\)] \ a-in \ a-\text{notin-rest} \\
  \item \have a \in \text{read-only } (\text{share ?take-sb } S) \quad \by \text{(auto simp add: aargh)} \\
  \item \from \text{read-only-share-unowned-in } [\OF \consis-take \this] \ a-\text{notin-shared} \\
  \item \show \?thesis \quad \by \text{auto} \\
  \end{itemize} 
\end{proof} 
\end{align*}

\quad \text{qed}
\}\n
\text{then show } ?\text{case} \quad \by \text{(auto simp add: t aargh)}
\quad \text{qed}

\text{lemma weak-sharing-consistent-preserves-distinct-share-all-until-volatile-write:}
\begin{align*}
  \forall S. \ [\text{ownership-distinct ts; read-only-unowned } S \text{ ts;weak-sharing-consis ts;}
  \ i < \text{length ts}; ts!i = (p,\text{is},\emptyset, sb, D, O, R)] \\
  \implies \text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab} ) \text{ sb}) \quad O \cap \\
  \text{read-only } (\text{share-all-until-volatile-write ts } S) = \{} 
\end{align*}
\begin{proof} \quad \text{(induct ts)} 
\begin{itemize}
  \item \case \text{Nil} \quad \text{thus } ?\text{case} \quad \by \text{simp}
  \item \next \quad \text{case } \text{Cons } t \text{ ts} \quad \by \text{(auto simp add: t aargh)}
\end{itemize} 
\end{proof}

\text{next}
\begin{itemize}
  \item \text{case } \text{Cons } t \text{ ts} \quad \by \text{(auto simp add: t aargh)}
  \item \text{then interpret read-only-unowned } S \quad \text{t#ts}.
  \item \text{note i-bound = } i < \text{length } (t \# ts).
  \item \text{note ith = } (t \# ts)! i = (p,\text{is},\emptyset, sb, D, O, R).
\end{itemize}

\begin{align*}
  & \have \text{dist: ownership-distinct } (t\#ts) \quad \by \text{fact} \\
  & \text{then interpret ownership-distinct } t\#ts.
\end{align*}

\begin{itemize}
  \item \from \text{ownership-distinct-tl } [\OF \dist] \\
  \item \have \text{dist': ownership-distinct ts}.
\end{itemize}

\begin{align*}
  & \have \text{consis: weak-sharing-consis } (t\#ts) \quad \by \text{fact} \\
  & \text{then interpret weak-sharing-consis } t\#ts.
\end{align*}

\begin{itemize}
  \item \text{note consis' = weak-sharing-consis-tl } [\OF \consis]
\end{itemize}

\begin{align*}
  & \let ?\text{take-sb} = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab} ) \text{ sb}) \\
  & \let ?\text{drop-sb} = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab} ) \text{ sb}) 
\end{align*}
have aargh: \((\text{Not } \circ \text{is-volatile-Write}_{sb}) = (\lambda a. \neg \text{is-volatile-Write}_{sb} a)\)
  by (rule ext) auto
show ?case
proof (cases i)
case 0
from read-only-unowned [of 0] ith 0
have owns-ro: \(\mathcal{O} \cap \text{read-only } \mathcal{S} = \{\}\)
  by force
from weak-sharing-consis [of 0] ith 0
have weak-sharing-consistent \(\mathcal{O} \text{ sb}\)
  by force
with weak-sharing-consistent-append [of \(\mathcal{O} \text{ ?take-sb} \text{ ?drop-sb}\)]
have consis-take: weak-sharing-consistent \(\mathcal{O} \text{ ?take-sb}\)
  by auto
from weak-sharing-consistent-preserves-distinct [OF this owns-ro]
have dist-t: acquired True \(\text{?take-sb } \mathcal{O} \cap \text{read-only } (\text{share } ?\text{take-sb } \mathcal{S}) = \{\}\).
from read-only-shared-all-untill-volatile-write-subset [OF dist 'consis', of (share ?take-sb \(\mathcal{S}\))]
have ro-rest: read-only (share-all-until-volatile-write ts (share ?take-sb \(\mathcal{S}\))) \subseteq
  read-only (share ?take-sb \(\mathcal{S}\)) \cup
  \(\bigcup ( (\lambda (-, -, -, sb, -, \mathcal{O}, -). \mathcal{O} \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})) \cup \text{set ts})\))
  by auto

{}
fix a
assume a-in-sb: \(a \in \text{acquired True ?take-sb } \mathcal{O}\)
assume a-in-ro: \(a \in \text{read-only } (\text{share-all-until-volatile-write ts } (\text{share } ?\text{take-sb } \mathcal{S}))\)
have False
proof –

from Set.in-mono [rule-format, OF ro-rest a-in-ro] dist-t a-in-sb

  have a \(\in (\bigcup ((\lambda(-, -, -, sb, -, \mathcal{O}, -). \mathcal{O} \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})) \cup \text{set ts}))\)
    by auto
  then obtain j \(p_j i_s j \hat{d}_j s_b j \mathcal{D}_j \mathcal{O}_j \mathcal{R}_j\)
    where j-bound: \(j < \text{length ts and ts-j: ts}[j] = (p_j,i_s,j,\hat{d}_j,s_b,j,\mathcal{D}_j,\mathcal{O}_j,\mathcal{R}_j)\)
    and a-in-j: \(a \in \mathcal{O}_j \cup \text{all-acquired } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb})\)
    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 ?take-sb \(\mathcal{O}\)] a-in-sb all-acquired-append [of ?take-sb ?drop-sb]
  have a \(\in \mathcal{O} \cup \text{all-acquired sb}\)
    by auto

362
with a-in-j all-acquired-append [of (takeWhile (Not ◦ is-volatile-Write sbj) sbj)]
(dropWhile (Not ◦ is-volatile-Write sbj) sbj)]
dist
have False by fastforce
thus ?thesis ..
qed
}
then show ?thesis
using 0 ith
by (auto simp add: aargh)

next

case (Suc k)
from i-bound Suc have k-bound: k < length ts
by auto
from ith Suc have kth: ts!k = (p, is, θ, sb, D, O, R)
by auto

obtain p_t is_t O_t R_t D_t θ_t sb_t
where t: t=(p_t,is_t,θ_t,was_t,D_t,O_t,R_t)
by (cases t)

let ?take-sb_t = (takeWhile (Not ◦ is-volatile-Write sb_t) sb_t)
let ?drop-sb_t = (dropWhile (Not ◦ is-volatile-Write sb_t) sb_t)

have ro-unowned': read-only-unowned (share ?take-sb_t S) ts
proof
  fix j
  fix p_j is_j O_j R_j D_j θ_j sb_j
  assume j-bound: j < length ts
  assume jth: ts!j = (p_j, is_j, θ_j, sb_j, D_j, O_j, R_j)
  show O_j ∩ read-only (share ?take-sb_t S) = {}
  proof
    from read-only-unowned [of Suc j] j-bound jth
    have dist: O_j ∩ 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 ?take-sb_t ?drop-sb_t]
    have consis-t: weak-sharing-consistent O_t ?take-sb_t
      by auto
    { fix a
      assume a-in-j: a ∈ O_j
      assume a-ro: a ∈ read-only (share ?take-sb_t S)
      have False
      proof
        from a-in-j ownership-distinct [of 0 Suc j] j-bound t jth
        have (O_j ∪ all-acquired sb_j) ∩ (O_j ∪ all-acquired sb_j) = {}
        by fastforce
      }
with a-in-j all-acquired-append [of ?take-sb \?drop-sb]
have a \notin (\cal O_t \cup all-acquired ?take-sb)
by fastforce
from read-only-share-unowned [OF consis-t this a-ro]
have a \in read-only \cal 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 \cal S ts
assumes ro-unowned: read-only-unowned \cal S ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, \theta, sb, D, O, R)
assumes a-unacquired-others: \forall j < length ts. i \neq j \rightarrow
(let (\cdot, \cdot, sb_j, \cdot, \cdot) = ts!j in
a \notin all-acquired (takeWhile (Not \circ is-volatile-Write sb) sb_j))
assumes a-ro-share: a \in read-only (share sb \cal S)
shows a \in read-only (share (dropWhile (Not \circ is-volatile-Write sb) sb)
(share-all-until-volatile-write ts \cal S))
proof –
from consis
interpret sharing-consis \cal S ts .
interpret read-only-unowned \cal S ts by fact
from sharing-consis [OF i-bound ts-i]
have consis-sb: sharing-consistent \cal S \cal O sb.
from sharing-consistent-weak-sharing-consistent [OF this]
have weak-consis: weak-sharing-consistent \cal O sb.
from read-only-unowned [OF i-bound ts-i]
have owns-ro: \cal O \cap read-only \cal 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 \cal S \land a \notin all-acquired sb.
moreover
let ?take-sb = (takeWhile (Not \circ is-volatile-Write sb) sb)
let ?drop-sb = (dropWhile (Not ◦ 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 ∈ read-only (share sb Map.empty)
     with share-append [of ?take-sb ?drop-sb]
     have a-in': a ∈ read-only (share ?drop-sb (share ?take-sb Map.empty))
       by auto
     have owns-empty: O ∩ read-only Map.empty = {}
       by auto

     from weak-sharing-consistent-preserves-distinct [OF weak-consis-take owns-empty]
     have acquired True ?take-sb O ∩ read-only (share ?take-sb Map.empty) = {}. 

     from read-only-share-all-acquired-in [OF this weak-consis' a-in']
     have a ∈ read-only (share ?drop-sb Map.empty) ∨ a ∈ read-only (share ?take-sb Map.empty) ∧ a ∉ all-acquired ?drop-sb.
     moreover
     {
       assume a-ro-drop: a ∈ read-only (share ?drop-sb Map.empty)
       have read-only Map.empty ⊆ 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 ∈ read-only (share ?take-sb Map.empty)
       assume a-unacq-drop: a ∉ all-acquired ?drop-sb
       from read-only-share-unowned-in [OF weak-consis-take a-ro-take]
       have a ∈ O ∪ all-acquired ?take-sb by auto
       hence a ∈ O ∪ 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
   }

365
with all-acquired-append [of ?take-sb ?drop-sb]

obtain a \notin all-acquired ?take-sb and a \notin all-acquired ?drop-sb
by auto

with a-unacquired-others i-bound ts-i
have a-unacq: \( \forall j < \text{length ts.} \)
\[ (\text{let } (\mathbf{-}, \mathbf{-}, \mathbf{sb}_j, \mathbf{-}, \mathbf{-}, \mathbf{-}) = ts!j \text{ in} \) \]
a \notin all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \mathbf{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 \in read-only (share-all-until-volatile-write ts S).

from weak-consis weak-sharing-consistent-append [of \( \_ \) ?take-sb ?drop-sb]
have weak-consis-drop: weak-sharing-consistent (acquired True ?take-sb \( \_ \)) ?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 \( \_ \) \cap
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 \in all-acquired (dropWhile P \mathbf{sb}) \implies x \in all-acquired \mathbf{sb}
using all-acquired-append [of takeWhile P \mathbf{sb} dropWhile P \mathbf{sb}]
by auto

lemma all-acquired-takeWhile-in: x \in all-acquired (takeWhile P \mathbf{sb}) \implies x \in all-acquired \mathbf{sb}
using all-acquired-append [of takeWhile P \mathbf{sb} dropWhile P \mathbf{sb}]
by auto

lemmas all-acquired-takeWhile-dropWhile-in = all-acquired-takeWhile-in all-acquired-dropWhile-in

lemma split-in-read-only-reads:
\( \bigwedge O. a \in \text{read-only-reads } O \ x s \implies (\exists t v y s z. x s = y s @ \text{Read}_{sb} \text{False } a t v \# z s \land a \notin \text{acquired True } y s O) \)
proof (induct xs)
case Nil thus \(?\)case by simp

366
next
  case (Cons x xs)
  have a-in: a ∈ read-only-reads \( O (x#xs) \) 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 ∈ read-only-reads \( O \) xs
      by (auto simp add: Write_{sb} False)
      from Cons.hyps [OF this] obtain t v ys zs where
      xs: xs=ys@Read_{sb} 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} False a t v # zs a ∉ acquired True (x#ys) \( O \)
      by (simp add: Write_{sb} False)
      then show ?thesis
      by blast
    next
      case True
      from a-in have a ∈ read-only-reads \( (O ∪ A − R) \) xs
      by (auto simp add: Write_{sb} True)
      from Cons.hyps [OF this] obtain t v ys zs where
      xs: xs=ys@Read_{sb} 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} False a t v # zs a ∉ acquired True (x#ys) \( O \)
      by (simp add: Write_{sb} True)
      then show ?thesis
      by blast
    qed
  next
  case (Read_{sb} volatile a′ t′ v′)
  show ?thesis
  proof (cases ¬ volatile ∧ a ∉ \( O \) ∧ a′=a)
    case True
    with Read_{sb} show ?thesis
  by fastforce
  next
    case False
    with a-in have a ∈ read-only-reads \( O \) xs
    by (auto simp add: Read_{sb} split: if-split_asm)
    from Cons.hyps [OF this] obtain t v ys zs where
    xs: xs=ys@Read_{sb} 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} False a t v # zs a ∉ acquired True (x#ys) \( O \)
    by (simp add: Read_{sb})
    then show ?thesis
  qed
by blast 
qed

next
case "Prog"sb
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@Readsb False a t v # zs and a-notin: a /∈ acquired True ys O
by auto
with xs a-notin obtain x#xs=(x#ys)@Readsb False a t v # zs a /∈ acquired True (x#ys) O
by (simp add: Progsb)
then show ?thesis
by blast

next
case ("Ghost"sb 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@Readsb 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)@Readsb False a t v # zs a /∈ acquired True (x#ys) O
by (simp add: Ghostsb)
then show ?thesis
by blast
qed
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
    Readsb volatile a - - ⇒ (if a /∈ W ∧ ¬ volatile
    then insert a (unforwarded-non-volatile-reads xs W)
    else (unforwarded-non-volatile-reads xs W))
  | Write - a - - - - - - ⇒ unforwarded-non-volatile-reads xs (insert a W)
  | - ⇒ unforwarded-non-volatile-reads xs W)

lemma unforwarded-non-volatile-reads-non-volatile-Readsb:
∀W. unforwarded-non-volatile-reads sb W ⊆ outstanding-refs is-non-volatile-Readsb sb
apply (induct sb)
apply (auto split: memref.splits if-split-asm)
lemma in-unforwarded-non-volatile-reads-non-volatile-Read\_sb:
\[
a \in \text{unforwarded-non-volatile-reads sb } W \implies a \in \text{outstanding-refs is-non-volatile-Read}\_sb\ sb
\]
using unforwarded-non-volatile-reads-non-volatile-Read\_sb
by blast

lemma unforwarded-non-volatile-reads-antimono:
\[
\forall W \quad W \subseteq W' \implies \text{unforwarded-non-volatile-reads xs } W \subseteq \text{unforwarded-non-volatile-reads xs } W'
\]
apply (induct xs)
apply (auto split: memref.splits dest: insert-monoD)
done

lemma unforwarded-non-volatile-reads-antimono-in:
\[
x \in \text{unforwarded-non-volatile-reads xs } W \implies W \subseteq W'
\implies x \in \text{unforwarded-non-volatile-reads xs } W
\]
using unforwarded-non-volatile-reads-antimono
by blast

lemma unforwarded-non-volatile-reads-append:
\[
\forall W. \quad \text{unforwarded-non-volatile-reads } (xs@ys) W =
(\text{unforwarded-non-volatile-reads xs } W \cup
\text{unforwarded-non-volatile-reads ys } (W \cup \text{outstanding-refs is-Write}\_sb\ xs))
\]
apply (induct xs)
apply clarsimp
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. \quad m' a = m a
assumes subset: unforwarded-non-volatile-reads sb W \subseteq A
assumes consis-m: \text{reads-consistent pending-write } O \ m \ sb
shows \text{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 \cases by simp
next
case (Cons r sb)
note mem-eq = \forall a \in A \cup W. \quad m' a = m a
note subset = \text{unforwarded-non-volatile-reads } (r#sb) \ W \subseteq A
note consis-m = \text{reads-consistent pending-write } O \ m \ (r#sb)

show \cases
proof (cases r)
case (Write\_sb\ volatile a sop v A' L R W')
from subset obtain
\[\text{subset': unforwarded-non-volatile-reads sb } (\text{insert a } W) \subseteq A\]
by (auto simp add: Write\sb)  
from mem-eq  
have mem-eq\':  
\[ \forall a' \in (A \cup (\text{insert } a \ W)). \ (m'(a:=v)) \ a' = (m(a:=v)) \ a' \]  
by (auto)  
show ?thesis  
proof (cases volatile)  
case True  
from consis-m obtain  
consis\': reads-consistent True \( (O \cup A' - R) \ (m(a := v)) \) sb and  
no-volatile-Read\sb\': outstanding-refs is-volatile-Read\sb sb = {}  
by (simp add: Write\sb True)  
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.  
with no-volatile-Read\sb  
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)  
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\ volatile a t v)  
show ?thesis  
proof (cases a \in W)  
case False  
from subset obtain  
subset\': unforwarded-non-volatile-reads sb W \subseteq A  
using False  
by (auto simp add: Read\sb\ True split: if-split-asm)  
from Cons.hyps [OF mem-eq\' subset' consis\' ]

370
show \( ?\text{thesis} \)
  by (simp add: \( \text{Read}_{ab} \) True)
next
case True
from subset have
  subset': unforwarded-non-volatile-reads \( sb \) \( W \subseteq \)
    insert \( a \) \( A \)
  using True
  apply (auto simp add: \( \text{Read}_{ab} \) volatile split: if-split-asm)
done
from mem-eq True have mem-eq': \( \forall a' \in (\text{insert} \ a \ A) \cup W \). \( m' a' = m a' \)
  by auto
from Cons.hyps [OF mem-eq' subset' consis'] show \( ?\text{thesis} \)
  by (simp add: \( \text{Read}_{ab} \) 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 \( \lor a \in O \) \( \rightarrow v=m \ a \))
  by (simp add: \( \text{Read}_{ab} \) False)
show \( ?\text{thesis} \)
  proof (cases \( a \in W \))
next
case True
from mem-eq subset \( \text{Read}_{ab} \) True non-vol have \( m' a = m a \)
  by (auto simp add: False)
from subset obtain
  subset': unforwarded-non-volatile-reads \( sb \) \( W \subseteq \) insert \( a \) \( A \)
  using False
  by (auto simp add: \( \text{Read}_{ab} \) non-vol split: if-split-asm)
from mem-eq True have mem-eq': \( \forall a' \in (\text{insert} \ a \ A) \cup W \). \( m' a' = m a' \)
  by auto
with Cons.hyps [OF mem-eq' subset' consis'] v show \( ?\text{thesis} \)
  by (simp add: \( \text{Read}_{ab} \) non-vol)
next
case False
from mem-eq subset \( \text{Read}_{ab} \) False non-vol have meq: \( m' a = m a \)
  by (clarsimp )
from subset obtain
  subset': unforwarded-non-volatile-reads \( sb \) \( W \subseteq A \)
  using non-vol False
  by (auto simp add: \( \text{Read}_{ab} \) non-vol split: if-split-asm)
from mem-eq non-vol have mem-eq': \( \forall a' \in A \cup W \). \( m' a' = m a' \)
  by auto
with Cons.hyps [OF mem-eq' subset' consis'] v meq show \( ?\text{thesis} \)
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 ∪ 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) ⊆ 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} split: if-split-asm)
  from subset-acq obtain
    a-A: \(\neg\) volatile \(\longrightarrow a \in O \longrightarrow a \in A\) and
    subset-acq': acquired-reads True (takeWhile (Not 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)

373
from Cons.hyps [OF mem-eq' subset' subset-acq' consis']
show ?thesis
by (simp add: Read$_{sb}$ True)

next
case False
note non-vol = this
dfrom consis-m obtain

consis': reads-consistent False $O$ m sb and
v: a $\in$ $O$ $\longrightarrow$ v=m a
by (simp add: Read$_{sb}$ False)

from mem-eq a-A v have v': a $\in$ $O$ $\longrightarrow$ v'=m' a
by (auto simp add: non-volatile)

from Cons.hyps [OF mem-eq' subset' subset-acq' consis'] v'
show ?thesis
by (simp add: Read$_{sb}$ False)

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:$\forall S$ $O$.
[non-volatile-owned-or-read-only True $S$ $O$ sb;
 a $\in$ read-only-reads $O$ sb]
$\Longrightarrow$
$\exists$ xs ys t v. sb=xs@ Read$_{sb}$ False a t v # ys $\land$ a $\in$ read-only (share xs $S$) $\land$ a $\notin$
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

from Cons.prems obtain
a-ro: a $\in$ read-only-reads ($O$ $\cup$ A $-$ R) sb and
nvo': 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)

374
from Cons.hyps [OF nvo' a-ro]

obtain xs ys t v where
sb = xs @ Read_{ab} 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: Write_{sb} True)
done

next
  case False
  from Cons.prems obtain
a-ro: a ∈ read-only-reads O sb and
nvo': non-volatile-owned-or-read-only True S O sb
by (clarsimp simp add: Write_{sb} False)

  from Cons.hyps [OF nvo' a-ro]
  obtain xs ys t v where
sb = xs @ Read_{ab} False a t v # ys ∧ a ∈ read-only (share xs S) ∧ a /∈ read-only-reads O 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: Write_{sb} False)
done

qed

next
  case (Read_{ab} volatile a' t v)
  proof ?thesis
    case (cases a'=a ∧ a /∈ O ∧ ¬ volatile)
    show True
      with Cons.prems have a ∈ read-only S
      by (simp add: Read_{ab})

      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 ∈ 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'a-ro]
  obtain xs ys t' v' where
  sb = xs @ Read sb False a t' v' # ys ∧ a ∈ read-only (share xs S) ∧ a /∈ read-only-reads O 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 ∈ read-only-reads O 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 ∧ a ∈ read-only (share xs S) ∧ a /∈ read-only-reads O 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: Prog sb)
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 ⊕ R ⊕ A L) (O ∪ A – R) sb

376
by (clarsimp simp add: Ghost_{sb})

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 ⊆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_{sb})
done

qed

lemma read-only-read-acquired-witness: \( \forall S, O. \)
  \( \neg\neg\text{non-volatile-owned-or-read-only True } S, O, \text{sb; sharing-consistent } S, O, \text{sb;}
  \text{a /\in read-only } S; \text{ a /\in } O; \text{ a \in read-only-reads } O, \text{sb} \)
  \implies \exists xs, ys, t, v. \text{sb} = \text{xs@ Read}_{sb} \text{False a t v ≠ ys ∧ a \in all-acquired xs ∧ a \in read-only (share xs S) ∧}
  \text{a /\in read-only-reads } O, \text{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 ⊆L) (O ∪ A − R) sb and
  a-nro: a /\in read-only S and
  a-unowned: a /\in O and
  a-ro': a \in read-only-reads (O ∪ A − R) sb 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 ⊆L) (O ∪ A − R) sb
by (clarsimp simp add: Write_{sb} True)
from R-owns a-unowned
have a-R: a /\in R

377
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 @ Readₜb False a t v' # ys and
  ro: a ∈ read-only (share xs (S ⊕ W R ⊕ₜ A L)) and
  a-ro-xs: a /∈ read-only-reads (O ∪ A − 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: Writeₜb volatile)
done
next

case False
with a-unowned R-owns a-nro L-A A-R
obtain a-nro': a /∈ read-only (S ⊕ W R ⊕ₜ 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']
obtain xs ys t v' where sb = xs @ Readₜb False a t v' # ys ∧
  a ∈ all-acquired xs ∧ a ∈ read-only (share xs (S ⊕ W R ⊕ₜ A L)) ∧
  a /∈ read-only-reads (O ∪ A − 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: Writeₜb volatile)
done

next
  qed

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
a' ∈ O and
nvo': non-volatile-owned-or-read-only True S O sb
by (clarsimp simp add: Writeₜb False)
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 ∈ all-acquired xs ∧ a ∈ read-only (share xs S) ∧ a /∈ 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: 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
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 @ Read$_{sb}$ False a t v' # ys ∧
a ∈ all-acquired xs ∧ a ∈ read-only (share xs S) ∧ a /∈ 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$_{sb}$)
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
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 ∧
  a ∈ all-acquired xs ∧ a ∈ read-only (share xs S) ∧ a /∈ 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: Prog_{sb})
done

next

case (Ghost_{sb} 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-ro': a ∈ read-only-reads (O ∪ A − R) sb 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})

from R-owns a-unowned
have a-R: a /∈ 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 ⊕ W R ⊆_A L)) and
  a-ro-xs: a /∈ read-only-reads (O ∪ A − 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: 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 \oplus Read_{sb} False a t v' \# ys \land
a \in all-acquired xs \land a \in read-only (share xs (S \oplus W R \ominus A L)) \land
a \notin read-only-reads (O \cup A - 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: Ghost_{sb})
done
qed
qed

lemma unforwarded-not-written: \forall W. a \in unforwarded-non-volatile-reads sb W \implies a \notin W
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')
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 ?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 Progs_{sb}
with Cons.hyps [of W] Cons.prems show ?thesis
by (auto split: if-split_asm)
case Ghost_{sb}

with Cons.hyps [of W] Cons.prems show ?thesis
  by (auto split: if-split-asm)
qed
qed

lemma unforwarded-witness;\(\forall X.\)
\[ a \in \text{unforwarded-non-volatile-reads } sb \ X \]

\[ \Rightarrow \exists xs \ ys \ t \ v. sb=xs@ Read_{sb} \text{ False } a \ t \ v \ # \ ys \ \& \ a \notin \text{outstanding-refs is-Write}_{sb} \ xs \]

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
      from Cons.prems obtain
        a-unforw: a \in \text{unforwarded-non-volatile-reads } sb \ (\text{insert } a' \ X)
      by (clarsimp simp add: Write_{sb} True)
      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 \at \ Read_{sb} \text{ False } a \ t \ v \ # \ ys \ \&
        a \notin \text{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 \in \text{unforwarded-non-volatile-reads } sb \ (\text{insert } a' \ X)
      by (clarsimp simp add: Write_{sb} False)
  next

382
from unforwarded-not-written [OF a-unforw]

have a’ ≠ 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 ∉ 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 False)
done

c qed

next

apply (cases a’ ≠ a ∧ a ∉ X ∧ ¬ volatile)
case True

show ?thesis

proof (cases a’ = a ∧ a ∉ 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

apply (cases a’ ≠ a ∧ a ∉ X ∧ ¬ volatile)
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 @ Read(sb False a t v # ys ∧
a ∉ outstanding-refs is-Write(sb 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 sb)
done
qed
next
case Prog sb
from Cons.prems obtain a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Prog sb)

from Cons.hyps [OF a-unforw]
obtain xs ys t v where
sb = xs @ Read sb False a t v # ys ∧
a /∈ outstanding-refs is-Write sb 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: Prog sb)
done

next
case (Ghost sb A L R W)
from Cons.prems obtain a-unforw: a ∈ unforwarded-non-volatile-reads sb X
by (auto simp add: Ghost sb)

from Cons.hyps [OF a-unforw]
obtain xs ys t v where
sb = xs @ Read sb False a t v # ys ∧
a /∈ outstanding-refs is-Write sb 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: ∀S O X.
[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;
a ∈ unforwarded-non-volatile-reads sb X]
⇒
∃xs ys t v. sb=x@ Read sb False a t v # ys ∧ a ∈ all-acquired xs ∧
a /∈ 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
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-ro′: a ∈ read-only-reads (O ∪ A − R) sb 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 and
a-unforw: a ∈ unforwarded-non-volatile-reads sb (insert a′ X)
by (clarsimp simp add: Write sb True)

from unforwarded-not-written [OF a-unforw]
have a-notin: a /∈ insert a′ X.
from R-owns a-unowned
have a-R: a /∈ R
by auto
show ?thesis
proof (cases a ∈ A)
case True

from unforwarded-witness [OF a-unforw]
obtain xs ys t v′ where
sb: sb = xs @ Read sb False a t v′ # ys and
a-x: a /∈ outstanding-refs is-Write sb xs
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

385
next

\textbf{case} False

\textbf{with} a-unowned R-owns a-nro L A- R

\textbf{obtain} a-nro': a \notin \text{read-only} (S \oplus W \ominus A L) \textbf{and} a-unowned; a \notin O \cup A - R

\textbf{by} (force simp add: in-read-only-convs)

\textbf{from} Cons.hyps [OF nvo' consis' a-nro' a-unowned' a-ro' a-unforw]

\textbf{obtain} xs ys t v' \textbf{where} sb = xs @ \text{Read}_{sb} False a t v' \# ys ∧

\qquad a \in \text{all-acquired} xs ∧

\qquad a \notin \text{outstanding-refs} is-\text{Write}_{sb} xs

\textbf{by} blast

\textbf{with} a-notin \textbf{show} ?thesis

\textbf{apply} –

\textbf{apply} (rule-tac x=x#xs in exI)

\textbf{apply} (rule-tac x=ys in exI)

\textbf{apply} (rule-tac x=t in exI)

\textbf{apply} (rule-tac x=v' in exI)

\textbf{apply} (clarsimp simp add: Write sb False split: if-split-asm)

\textbf{done}

\textbf{qed}

\textbf{next}

\textbf{case} False

\textbf{from} Cons.prems \textbf{obtain}

\textbf{consis}'\textbf{: sharing-consistent} S O sb \textbf{and}

\textbf{a-nro}'\textbf{: a \notin \text{read-only} S} \textbf{and}

\textbf{a-unowned: a \notin O} \textbf{and}

\textbf{a-ro}'\textbf{: a \in \text{read-only-reads} O sb} \textbf{and}

\textbf{a' \in O} \textbf{and}

\textbf{nvo}'\textbf{: non-volatile-owned-or-read-only} True S O sb \textbf{and}

\textbf{a-unforw'}\textbf{: a \in \text{unforwarded-non-volatile-reads} sb (insert a' X)}

\textbf{by} (auto simp add: Write sb False split: if-split-asm)

\textbf{from} unforwarded-not-written [OF a-unforw]

\textbf{have} a-notin: a \notin \text{insert} a' X.

\textbf{from} Cons.hyps [OF nvo' consis' a-nro' a-unowned a-ro' a-unforw']

\textbf{obtain} xs ys t v' \textbf{where}

\qquad sb = xs @ \text{Read}_{sb} False a t v' \# ys ∧

\qquad a \in \text{all-acquired} xs ∧ a \notin \text{outstanding-refs} is-\text{Write}_{sb} xs

\textbf{by} blast

\textbf{with} a-notin \textbf{show} ?thesis

\textbf{apply} –

\textbf{apply} (rule-tac x=x#xs in exI)

\textbf{apply} (rule-tac x=ys in exI)

\textbf{apply} (rule-tac x=t in exI)

\textbf{apply} (rule-tac x=v' in exI)

\textbf{apply} (clarsimp simp add: Write sb False)

\textbf{done}
qed

next
case (Read_{sb} volatile a' t v)
from Cons.prems
obtain
  consis': sharing-consistent \( S \setminus O \) \( sb \) 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 \( S \setminus O \) \( sb \) and
  a-unforw: a \( \in \) 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 \( \in \) all-acquired xs ∧ a \( \notin \) 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 \( S \setminus O \) \( sb \) 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 \( S \setminus O \) \( sb \) 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-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

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: Progf)
done

next
case (Ghostf 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-ro': a ∈ read-only-reads (O ∪ A − R) sb 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 and
a-unforw: a ∈ unforwarded-non-volatile-reads sb (X)
by (clarsimp simp add: Ghostf)

from unforwarded-not-written [OF a-unforw]
have a-notin: a /∈ X.
from R-owns a-unowned
have a-R: a /∈ R
  by auto
show ?thesis
proof (cases a ∈ A)
case True

from unforwarded-witness [OF a-unforw]
obtain xs ys t v' where
sb: sb = xs @ Readf False a t v' # ys and
a-xs: a /∈ outstanding-refs is-Writef xs
  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: Ghostf)
done

next
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 @ Readf False a t v' # ys ∧
a ∈ all-acquired xs ∧
a /∈ outstanding-refs is-Writef 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 sb)
done
qed
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 ∈ read-only S → a ∈ read-only S'
  assumes a-in: a ∈ read-only (S ⊕ W R)
  shows a ∈ read-only (S' ⊕ W R)
  using ro a-in
  by (auto simp add: in-read-only-convs)
lemma
  assumes ro: a ∈ read-only S → a ∈ read-only S'
  assumes a-in: a ∈ read-only (S ⊕ A L)
shows a ∈ read-only (S′ ⊕ 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:
\[ S S' O \text{ pending-write.} \]
\[ \text{∀ } a ∈ \text{read-only-reads } O \text{ sb. } a ∈ \text{read-only } S \longrightarrow a ∈ \text{read-only } S' \]
\[ \implies \text{non-volatile-owned-or-read-only pending-write } S' O \text{ sb} \]
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write \_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
ro’: ∀ a read-only-reads (O ∪ A − R) sb. a ∈ read-only S → a ∈ read-only S’
by (clarsimp simp add: Write_\_volatile)
from ro’
have ro’’: ∀ a read-only-reads (O ∪ A − R) sb.
a ∈ read-only (S ⊕ W R ⊕ A L) → a ∈ read-only (S’ ⊕ W R ⊕ A L)
by (auto simp add: in-read-only-convs)
from Cons.prems [OF nvo’ ro’’]
show ?thesis
by (clarsimp simp add: Write_\_volatile)
next
case False
with Cons.prems [of pending-write S O S’] Cons.prems show ?thesis
by (auto simp add: Write_\_volatile)
qed
next
case (Read_\_volatile \_a t v)
show ?thesis
proof (cases volatile)
case True
with Cons.prems [of pending-write S O S’] Cons.prems show ?thesis
by (clarsimp simp add: Read_\_volatile)
next
case False
note non-vol = this
show ?thesis
proof (cases a ∈ O)
case True
with Cons.hyps \([\text{of pending-write } S \odot S']\) Cons.prems show \(?\text{thesis}\)
by (auto simp add: Read\_sb non-vol)
next
case False
from Cons.prems Cons.hyps \([\text{of pending-write } S \odot S']\) show \(?\text{thesis}\)
by (clarsimp simp add: Read\_sb non-vol False)
qed
qed
next
case Prog\_sb
with Cons.hyps \([\text{of pending-write } S \odot S']\) Cons.prems show \(?\text{thesis}\)
by (auto)
next
case (Ghost\_sb A L R W)
from Cons.hyps \([\text{of pending-write } (S \oplus W R \ominus A L) \odot A - R S' \oplus W R \ominus A L]\)
Cons.prems
show \(?\text{thesis}\)
by (auto simp add: Ghost\_sb in-read-only-convs)
qed
qed

lemma non-volatile-owned-or-read-only-read-only-reads-eq' :
\[
\bigwedge S S' O. \\
[\text{non-volatile-owned-or-read-only False } S \odot O]; \\
\forall a \in \text{read-only-reads } (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) sb) O) \\
(\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) sb). a \in \text{read-only } S \rightarrow a \in \text{read-only } S' \\
] \\
\implies \text{non-volatile-owned-or-read-only False } S' \odot O sb
\]
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
note volatile=\text{this}
from Cons.prems obtain 
\[\text{nvo'}; \text{non-volatile-owned-or-read-only True } (S \oplus W R \ominus A L) (O \cup A - R) sb \text{ and} \]
\[\text{ro'}; \forall a \in \text{read-only-reads } (O \cup A - R) sb. a \in \text{read-only } S \rightarrow a \in \text{read-only } S' \]
by (clarsimp simp add: Write\_sb volatile)
from ro'
have \[\text{ro''}; \forall a \in \text{read-only-reads } (O \cup A - R) sb. \]
a \in read-only \((S \oplus W R \ominus A L) \rightarrow a \in 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 ?thesis

by (clarsimp simp add: Write_{sb} volatile)

next

case False

with Cons.hyps [of \S \S'] Cons.prems show ?thesis

by (auto simp add: Write_{sb})

qed simp: Write_{sb}

done

next

case (Read_{sb} volatile a t v)

show ?thesis

proof (cases volatile)

with Cons.hyps [of \S \S'] Cons.prems show ?thesis

by (auto simp add: Read_{sb})

next

case False

note non-vol = this

show ?thesis

proof (cases a \in O)

case True

with Cons.hyps [of \S \S'] Cons.prems show ?thesis

by (auto simp add: Read_{sb} non-vol)

next

case False

from Cons.hyps [of \S \S'] Cons.prems show ?thesis

by (clarsimp simp add: Read_{sb} non-vol False)

qed

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.hyps [of (\S' \oplus W) \ominus A L) \circ \cup A \cup R \S' \ominus W R \ominus A L] Cons.prems show ?thesis

by (auto simp add: Ghost_{sb} in-read-only-convs)

qed

qed

lemma no-write-to-read-only-memory-read-only-reads-eq:

\forall S S'.
[no-write-to-read-only-memory S sb;
 \forall a \in outstanding-refs is-Write_{sb} sb. a \in read-only S' \rightarrow a \in read-only S
]

\implies no-write-to-read-only-memory S' sb

proof (induct sb)

case Nil thus ?case by simp
lemma reads-consistent-drop:
reads-consistent False \( \mathcal{O} \) \( m \) \( \text{sb} \)
\[ \Rightarrow \] reads-consistent True
\[(\text{acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{ab} \sb) \mathcal{O})})\]
\[(\text{flush (takeWhile (Not \circ \text{is-volatile-Write}_{ab} \sb) \text{m})})\]
\[(\text{dropWhile (Not \circ \text{is-volatile-Write}_{ab} \sb)})\]
using reads-consistent-append [of False $O$ m (takeWhile (Not $\circ$ is-volatile-Write$_{ab}$) sb)]
apply (cases outstanding-refs is-volatile-Write$_{ab}$ 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 (Not $\circ$ is-volatile-Write$_{ab}$) sb))
apply (fastforce simp add: outstanding-refs-conv)
apply (frule dropWhile-ConsD)
apply (clarsimp split: memref.splits)
done

lemma outstanding-refs-non-volatile-Read$_{ab}$-all-acquired-dropWhile':
\[
\forall m. S \ O \ pending-write .
\quad \begin{array}{l}
\text{[reads-consistent pending-write } O \ m \ sb; \text{non-volatile-owned-or-read-only pending-write}
\quad S \ O \ sb; \\
\text{a } \in \text{ outstanding-refs is-non-volatile-Read$_{ab}$ (dropWhile (Not } \circ \text{ is-volatile-Write$_{ab}$) sb)]}
\Rightarrow a \in O \lor a \in \text{all-acquired sb } \lor
\quad a \in \text{read-only-reads (acquired True (takeWhile (Not } \circ \text{is-volatile-Write$_{ab}$) sb) } O )
\quad (\text{dropWhile (Not } \circ \text{is-volatile-Write$_{ab}$) sb})
\end{array}
\]
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
note volatile=this
from Cons.prems obtain
non-vo: non-volatile-owned-or-read-only True (S $\oplus_W$ R $\ominus_A$ L)
(\text{O } \cup \text{ A }\ominus R) \text{ sb and}
out-vol: outstanding-refs is-volatile-Read$_{ab}$ sb = { } and
out: a $\in$ outstanding-refs is-non-volatile-Read$_{ab}$ sb
by (clarsimp simp add: Write$_{ab}$ True)
show ?thesis
proof (cases a $\in$ O)
case True
show ?thesis
by (clarsimp simp add: Write$_{ab}$ True volatile)
next
case False
from outstanding-non-volatile-Read$_{ab}$-acquired-or-read-only-reads [OF non-vo out]
have a-in: a $\in$ acquired-reads True sb (O $\cup$ A $\ominus$ R) $\lor$
\quad a $\in$ read-only-reads (O $\cup$ A $\ominus$ R) sb
by auto
with acquired-reads-all-acquired [of True sb (O U A - R)]
show ?thesis
  by (auto simp add: Write sb volatile)
qed

next
  case False
  with Cons 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 Prog sb
  with Cons show ?thesis
  by (auto simp add: Read sb)
next
  case (Ghost sb A L R W)
  with Cons.hyps [of pending-write O U A - R m S @W R @A L]
read-only-reads-antimono [of O O U A - R]
  Cons.prems show ?thesis
  by (auto simp add: Ghost sb)
qed
qed

end

theory ReduceStoreBufferSimulation
imports ReduceStoreBuffer
begin

locale initial sb = simple-ownership-distinct + read-only-unowned + unowned-shared +
constrains ts::{p, p store-buffer, bool.owns.rel} 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 sb \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!i = (p, is, is, 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


395
qed

sublocale initial \subseteq outstanding-volatile-writes-unowned-by-others
proof
fix i j p; is, O, R; D, \theta; sb, p; i; j; O, R; D, \theta; sb
assume i-bound: i < length ts and
j-bound: j < length ts and
neq-i-j: i \neq j and
ts-i: ts ! i = (p, is, \theta, sb, D, O, R) and
ts-j: ts ! j = (p, i; j, \theta, sb, D, O, R)
show (O \cup all-acquired sb) \cap outstanding-refs is-volatile-Write \subseteq \{\}
qed

sublocale initial \subseteq read-only-reads-unowned
proof
fix i j p; is, O, R; D, \theta; sb, p; i; j; O, R; D, \theta; sb
assume i-bound: i < length ts and
j-bound: j < length ts and
neq-i-j: i \neq j and
ts-i: ts ! i = (p, is, \theta, sb, D, O, R) and
ts-j: ts ! j = (p, i; j, \theta, sb, D, O, R)
show (O \cup all-acquired sb) \cap read-only-reads (acquired True
(takeWhile (Not o is-volatile-Write \subseteq O) O)
(dropWhile (Not o is-volatile-Write \subseteq sb) = \{\})
qed

sublocale initial \subseteq ownership-distinct
proof
fix i j p; is, O, R; D, \theta; sb, p; i; j; O, R; D, \theta; sb
assume i-bound: i < length ts and
j-bound: j < length ts and
neq-i-j: i \neq j and
ts-i: ts ! i = (p, is, \theta, sb, D, O, R) and
ts-j: ts ! j = (p, i; j, \theta, sb, D, O, R)
show (O \cup all-acquired sb) \cap (O \cup all-acquired sb) = \{\}
by auto
qed

sublocale initial \subseteq valid-ownership ..

sublocale initial \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 \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

396
using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ no-outstanding-write-to-read-only-memory
proof
  fix i is ORD θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts ! i = (p, is, θ, 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 ⊆ valid-sharing ..
sublocale initial sb ⊆ valid-ownership-and-sharing ..

sublocale initial sb ⊆ load-tmps-distinct
proof
  fix i is ORD θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts ! i = (p, is, θ, sb, D, O, R)
  show distinct-load-tmps is
  using empty-is [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ read-tmps-distinct
proof
  fix i is ORD θ sb p
  assume i-bound: i < length ts
  assume ts-i: ts ! i = (p, is, θ, sb, D, O, R)
  show distinct-read-tmps sb
  using empty-sb [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ load-tmps-read-tmps-distinct
proof
  fix i is ORD 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 ∩ read-tmps sb = {}
  using empty-sb [OF i-bound ts-i] empty-is [OF i-bound ts-i] by auto
qed

sublocale initial sb ⊆ load-tmps-read-tmps-distinct ..

sublocale initial sb ⊆ valid-write-sops
proof
  fix i is ORD θ 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 ORD 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 ..

proof

fix i is OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
show reads-consistent False "m sb by auto
qed

sublocale initial_{sb} ⊆ valid-reads

proof

fix i is OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
show empty-sb [OF i-bound ts-i] by auto

sublocale initial_{sb} ⊆ valid-history

proof

fix i is OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,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 OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
show data-dependency-consistent-instrs (dom θ) is
using empty-is [OF i-bound ts-i] by auto
next
fix i is OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
show empty-is [OF i-bound ts-i] empty-sb [OF i-bound ts-i] by auto

sublocale initial_{sb} ⊆ load-tmps-fresh

proof

fix i is OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
show load-tmps is ∩ (fst ` write-sops sb) = {} using empty-is [OF i-bound ts-i] by auto

sublocale initial_{sb} ⊆ enough-flushs

proof

fix i is OR D θ sb p
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
show outstanding-refs is-volatile-Write sb sb sb = {} using empty-sb [OF i-bound ts-i] by auto

sublocale initial_{sb} ⊆ valid-program-history

proof

fix i is OR D θ sb p sb1 sb2
assume i-bound: i < length ts
assume ts-i: ts!i = (p,is,0,0D,0O,R)
assume sb: sb = sb1 @ sb2
show ∃ isa. instrs sb2 @ is = isa @ prog-instrs sb2
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 ∅ sb p
assume i-bound: i < length ts
assume ts-i: tsli = (p,i,∅,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 × memory × shared ⇒
(p, unit,bool,owns,rels) thread-config list × memory × shared ⇒ bool

(¬ ~ - [60,60] 100)

where

[m = flush-all-until-volatile-write tsab mab;]
S = share-all-until-volatile-write tsab S_ab;
length tsab = length ts;
∀ i < length tsab.

let (p, i_ab, θ, sb, D_ab, O, R) = ts_ab li;
suspends = dropWhile (Not o is-volatile-Write_ab) sb
in ∃ is D. instrs suspends ⊗ i_ab = is ⊗ prog-instrs suspends ∧
D_ab = (D ∨ outstanding-refs is-volatile-Write_ab sb ≠ { }) ∧
tsli = (hd-prog p suspends, is,
θ |' (Dom θ − read-trmps suspends),(),
D, acquired True (takeWhile (Not o is-volatile-Write_ab) sb) O,
release (takeWhile (Not o is-volatile-Write_ab) sb) (Dom S_ab) R )

⇒

(tsab,mab,S_ab) ~ (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 ⇒
(p,p store-buffer,bool,owns,rels) thread-config list ⇒ bool

(¬ ~h - [60,60] 100)

where

[length ts = length ts_i;]
∀ i < length ts_i.

(∃ O' D' R').

let (p,i,θ, sb,D, O,R) = ts_i li in
tsli = (p,i,θ, filter is-Write_ab sb,D',O',R') ∧
(filter is-Write_ab sb = [] ➔ sb=[])

⇒

ts ~h ts

lemma (in initial_ab) history-refl:ts ~h ts
apply –
apply (rule sim-history-config,intros)
apply simp
apply clarsimp
subgoal for i
apply (case-tac tsli)
apply (drule-tac i=i in empty-sb)
apply assumption
apply auto
done
done

399
lemma share-all-empty: \( \forall p \; is \; xs \; sb \; D \; O \; R. \; i < length \; ts \implies ts!i = (p, is, xs, sb, D, O, R) \implies sb = [] \)
apply (induct ts)
apply clarsimp
apply (frule-tac x=0 in spec)
apply clarsimp
apply force
done

lemma flush-all-empty: \( \forall p \; is \; xs \; sb \; D \; O \; R. \; i < length \; ts \implies ts!i = (p, is, xs, sb, D, O, R) \implies sb = [] \)
apply (induct ts)
apply clarsimp
applyclarsimp
apply (frule-tac x=0 in spec)
apply clarsimp
apply force
done

lemma sim-config-emptyE:
assumes empty: \( \forall p \; is \; xs \; sb \; D \; O \; R. \; i < length \; ts_{sb} \implies ts_{sb}!i = (p, is, xs, sb, D, O, R) \implies sb = [] \)
assumes sim: \((ts_{ab}, m_{ab}, S_{ab}) \sim (ts, m, S)\)
shows \(S = S_{ab} \land m = m_{ab} \land length \; ts = length \; ts_{ab} \land \)
\((\forall i < length \; ts_{ab}. \)
let \((p, is, \emptyset, sb, D, O, R) = ts_{ab}!i \)
in \(ts!i = (p, is, \emptyset, (), D, O, R)\)
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 (clarsimp simp add: leq [rule-format])
apply clarsimp
apply assumption
apply (auto simp add: Let-def)
done
done
done
qed

lemma sim-config-emptyI:
assumes empty: \( \forall p \; is \; xs \; sb \; D \; O \; R. \; i < length \; ts_{sb} \implies ts_{sb}!i = (p, is, xs, sb, D, O, R) \implies sb = [] \)
assumes leq: \( length \; ts = length \; ts_{sb} \)
assumes ts: \((\forall i < length \; ts_{sb}. \)
let \((p, is, \emptyset, sb, D, O, R) = ts_{sb}!i \)
in \(ts!i = (p, is, \emptyset, (), D, O, R)\)
shows \((ts_{ab}, m_{ab}, S_{ab}) \sim (ts, m_{ab}, S_{ab})\)
apply (rule sim-config.intros)
apply (clarsimp simp add: flush-all-empty [OF empty])
apply (clarsimp simp add: share-all-empty [OF empty])
apply (clarsimp simp add: leq)
apply (clarsimp)
apply (frule (1) empty [rule-format])
using ts
apply (auto simp add: Let-def)
done

400
lemma mem-eq-un-eq: \[ \text{length } ts' = \text{length } ts; \forall i < \text{length } ts'. P \ (ts''!i) = Q \ (ts!i) \] \implies (\bigcup x \in \text{set } ts'. P x) = (\bigcup x \in \text{set } 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

lemma (in program) trace-to-steps:
assumes trace: trace c 0 k
shows steps: c 0 \implies_d^* c k
using trace
proof (induct k)
case 0
show c 0 \implies_d^* c 0
  by auto
next
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 \implies_d^* c k .
also
term program-trace
from prem interpret program-trace program-step c 0 Suc k .
from step [of k] have c (k) \implies_d c (Suc k)
  by auto
finally show ?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 0 \implies_d^* c l.
  with safe-reach c-0 show safe (c l)
    by (cases c l) (auto simp add: safe-reach-def)
  qed
  qed

lemma (in program-progress) safe-free-flowing-implies-safe-delayed':
  assumes init: initial_{\mathcal{S}_{1b}} t_{s_{1b}} S_{1b}
  assumes sim: (t_{s_{1b}} m_{s_{1b}}, S_{1b}) \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

401
from init
interpret ini: init_{ab} ts_{ab} S_{ab} .
from sim obtain
m: m = flush-all-until-volatile-write ts_{ab} m_{ab} and
S: S = share-all-until-volatile-write ts_{ab} S_{ab} and
leq: length ts_{ab} = length ts and
t-sim: \forall i < length ts_{ab},
  let (p, is_{ab}, \emptyset, sb, D_{ab}, O, R) = ts_{ab}!i;
  suspends = dropWhile (Not o is-volatile-Write_{ab}) sb
  in \exists D. instrs suspends \oplus is_{ab} = is \oplus prog-instrs suspends \land
  D_{ab} = (D \lor outstanding-refs is-volatile-Write_{ab} sb \neq \{\}) \land
  ts_i! = (hd-prog p suspends, i, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)
  acquired True (takeWhile (Not o is-volatile-Write_{ab}) sb) O,
  release (takeWhile (Not o is-volatile-Write_{ab}) sb) (dom S_{ab}) R )
by cases auto

from ini.empty-sb
have shared-eq: S = S_{ab}
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, R, D, \emptyset, sb, p_j, is_j, O_j, R_j, D_j, \emptyset, sb_j
assume i-bound: i < length ts and
  j-bound: j < length ts and
  neq-i-j: i \neq j and
  ts-i: ts_i! i = (p_i, is_i, \emptyset, sb_i, D_i, O_i, R_i) and
  ts-j: ts_j! j = (p_j, is_j, \emptyset, sb_j, D_j, O_j, 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_{ab}!i = (p_i, is_i, \emptyset, sb_i, D_i, O_i, 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_{ab}!j = (p_j, is_j, \emptyset, sb_j, D_j, O_j, 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 R D \emptyset sb p
assume i-bound: i < length ts
assume ts-i: ts_i! e = (p_i, is_i, \emptyset, sb, D, O, R)
show \{O\} \cap 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_{ab}!i = (p_i, is_i, \emptyset, sb, D_i, O_i, R_i)
  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
qed
have us: unowned-shared \( S \) \( ts \)
proof
  show \( \{- (\bigcup ((\lambda (\cdot, \cdot, \cdot, \cdot, O, \cdot). O) \cdot set ts)) \} \subseteq dom S \)
proof
  have \( \{- ((\lambda (\cdot, \cdot, \cdot, \cdot, O, \cdot). O) \cdot set ts)\} = \{- ((\lambda (\cdot, \cdot, \cdot, \cdot, O, \cdot). O) \cdot set ts)\} \)
    apply clarsimp
    apply (rule mem-eq-un-eq)
    apply (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
qed

\begin{proof}
fix \( i \) is \( O \ R D \emptyset \emptyset sb p \)
assume i-bound: \( i < \) length \( ts \)
assume \( ts\)-i: \( ts\)-i = \( (p, is, \emptyset, sb, D, O, R) \)
have \( R = \) 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
\end{proof}

with us have initial: initial \( (ts, m, S) \)
by (fastforce simp add: initial-def)

\begin{proof}
fix \( ts \) \( S \) \( m \) \( m' \)
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
\end{proof}
then show \( ?\)thesis
  by (clarsimp simp add: safe-reach-def)
qed

lemma map-onws-sb-owned: \( \forall j. j < \) length \( ts \) \( \Rightarrow \) map \( O\)-sb \( ts \) ! \( j = (O_j, sb_j) \) \( \Rightarrow \) 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

done

lemma map-onws-ownedsb-owned:∀j. j < length ts ⇒ O-sb (ts ! j) = (Oj, sbj) ⇒ owned (ts ! j) = Oj
apply (induct ts)
apply simp
subgoal for t ts j
apply (case-tac t)
apply auto
done
done

lemma read-only-read-acquired-unforwarded-acquire-witness:
∀S O X. non-volatile-owned-or-read-only True S O sb;
sharing-consistent S O sb: a \notin read-only S; a \notin O;
a \in unforwarded-non-volatile-reads sb X]
⇒⇒(\exists sop a’ v ys zs A L R W.
sb = ys \oplus Writeab True a’ sop v A L R W \neq zs ∧
a \in A ∧ a \notin outstanding-refs is-Writeab ys ∧ a’ ≠ a) ∨
(\exists A L R W ys zs. sb = ys \otimes Ghostab A L R W# zs ∧ a \in A ∧ a \notin outstanding-refs is-Writeab ys)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Writeab 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 read-only S and
a-unowned: a \notin A and
A-shared-owns: A \subseteq dom S ⊔ O and L-A: L ⊆ A and A-R: A \cap R = {} and
R-owns: R \subseteq O and
consis': sharing-consistent (S ⊕ W R ⊕ A L) (O ⊔ A − R) sb and
a-unforw: a \in unforwarded-non-volatile-reads sb (insert a’ X)
by (clarsimp simp add: Writeab True)
from unforwarded-not-written [OF a-unforw]
have a-notin: a \notin 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)

404
apply (rule-tac x=[] in exl)
apply (rule-tac x=sb in exl)
apply (simp add: Writeab volatile True a'-a)
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 nvo' consis' a-nro' a-unowned' a-unforw]
have (\exists \text{sop } a' \text{ v} ys zs A L R W.
  sb' = ys \oplus Writeab True a' \text{ sop } v A L R W \neq zs \land
  a \in A \land a' \notin \text{outstanding-refs is-Writeab } ys' \land a' \neq a) \lor
  (\exists A L R W ys zs. sb = ys \oplus Ghostab A L R W# zs \land a \in A \land a \notin \text{outstanding-refs is-Writeab } 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
    sb' = ys \oplus Writeab True a'' \text{ sop } v' A' L' R' W' \neq zs and
    props: a \in A' \land a' \notin \text{outstanding-refs is-Writeab } ys' \land a'' \neq a
  by auto

  show ?thesis
  using props False a-notin sb
  apply
  apply (rule disj1)
  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: Writeab volatile False a'-a)
done
next
assume ?ghst
then obtain ys zs A' L' R' W' where
  sb = ys \oplus Ghostab A' L' R' W' # zs and
  props: a \in A' \land a' \notin \text{outstanding-refs is-Writeab } ys
  by auto

  show ?thesis
  using props False a-notin sb
  apply
  apply (rule disj2)
  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: Writeab volatile False a'-a)
done
qed
qed
next

405
case False
  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' ∈ O and
  nvo': non-volatile-owned-or-read-only True S O sb and
  a-unforw': a ∈ 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 ∉ insert a' X.

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 sb True a' sop v A L R W ≠ zs ∧
  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 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 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 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 ?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 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

406
qed

qed

next

case (Read\sb\ volatile\ a' t v)

from Cons.prems

obtain

\text{consis': sharing-consistent } S O \sb \text{ and}

\text{a-nro': } a \notin \text{ read-only } S \text{ and}

\text{a-unowned: } a \notin O \text{ and}

\text{nvo': non-volatile-owned-or-read-only True } S O \sb \text{ and}

\text{a-unforw: } a \in \text{ unforwarded-non-volatile-reads } X \sb

by (auto simp add: Read\sb split: if-split-asm)

from Cons.hyps [OF nvo' consis' a-nro' a-unowned a-unforw]

have (\exists sop a' v ys zs A L R W.

\text{sb = ys } \oplus \text{ Write\sb True a' 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. 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: sb = ys } \oplus \text{ Write\sb True a'' sop v' A' L' R' W' } \# zs \text{ and}

\text{props: } a \in A' \land 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 exI)

apply (rule-tac x=a' in exI)

apply (rule-tac x=v' in exI)

apply (rule-tac x=(x#ys) in exI)

apply (rule-tac x=zs in exI)

apply (simp add: Read\sb)

done

next

assume ?ghst

then obtain ys zs A' L' R' W' where

\text{sb: sb = ys } \oplus \text{ Ghost\sb A' L' R' W' } \# zs \text{ and}

\text{props: } a \in A' \land 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 exI)

apply (rule-tac x=L' in exI)

apply (rule-tac x=R' in exI)

apply (rule-tac x=W' in exI)

apply (rule-tac x=(x#ys) in exI)

apply (rule-tac x=zs in exI)

apply (simp add: Read\sb)

done

qed

407
next
case Prog\textsubscript{sb}
from Cons.prems
obtain
\textit{consi}': sharing-consistent \(S O\) sb and
\textit{a-nro}': \(a \notin\) read-only \(S\) and
\textit{a-unowned}': \(a \notin\) \(O\) and
\textit{nvo}': non-volatile-owned-or-read-only True \(S O\) sb and
\textit{a-unforw}': \(a \in\) unforwarded-non-volatile-reads sb X
by (auto simp add: Prog\textsubscript{sb})
from Cons.hyps [OF nvo' consi' a-nro' a-unowned a-unforw]
have \((\exists\text{sop} a' v yz A L R W. sb = ys \oplus\text{Write}_{ab} True a' \text{sop} v A L R W \# z s \land a \in A \land a' \notin a) \lor \\
(\exists A L R W yz. sb = ys \oplus\text{Ghost}_{ab} A L R W \# z s \land a \in A \land a \notin \text{outstanding-refs is-Write}_{ab} ys)\)
(is \text{write} \lor \text{ghst})
by simp
then show \text{?thesis}
proof
assume \text{?write}
then obtain sop' a'' v' yz A' L' R' W' where
sb: sb = ys \oplus\text{Write}_{ab} True a'' \text{sop} v' A' L' R' W' \# z s and
props: a \in A' a' \notin \text{outstanding-refs is-Write}_{ab} ys \land a'' \neq a
by auto
show \text{?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\#yz in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Prog\textsubscript{sb})
done
next
assume \text{?ghst}
then obtain yz A' L' R' W' where
sb: sb = ys \oplus\text{Ghost}_{ab} A' L' R' W' \# z s and
props: a \in A' a \notin \text{outstanding-refs is-Write}_{ab} ys
by auto
show \text{?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\#yz in exl)
apply (rule-tac x=zs in exl)
apply (simp add: Prog\textsubscript{sb})
done
qed
next
case (\text{Ghost}_{ab} A L R W)

408
from Cons.prems obtain nvo': non-volatile-owned-or-read-only True \( (S \oplus W R \ominus A L) (O \cup A - R) \) sb and a-nro: a \( \notin \) read-only \( S \) and a-unowned: a \( \notin \) \( O \) 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 and a-unforw: a \( \in \) unforwarded-non-volatile-reads sb X by (clarsimp simp add: Ghost sb)

show ?thesis
proof (cases a \( \in A \))
case True then show ?thesis
apply --
apply (rule disjI2)
apply (rule-tac x=A in ex)
apply (rule-tac x=V in ex)
apply (rule-tac x=W in ex)
apply (rule-tac x=[] in ex)
apply (rule-tac x=nb in ex)
apply (simp add: Ghost sb True)
done
next
assume ?ghst then obtain \( \exists \) a unowned L-A R-owns a-nro L-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 \( \exists \) sop a' v y z a L R W, sb = y s @ Write \( \exists \) a' sop v A L R W # y z \( \wedge \)
 a \( \in A \) \( \wedge \) a \( \notin \) outstanding-refs is-Write \( \exists \) y s \( \wedge \) a' \( \neq \) a \( \vee \)
(\( \exists \) A L R W y z s b = y s @ Ghost \( \exists \) A L R W # y z \( \wedge \) a \( \in A \) \( \wedge \) a \( \notin \) outstanding-refs is-Write \( \exists \) y s)
(is ?write \( \vee \) ?ghst)
by simp
then show ?thesis
proof
assume ?write
then obtain sop' a'' v' y z a' L' R' W' where
sb: sb = y s @ Write \( \exists \) a'' sop' v' A' L' R' W' # y z and
props: a \( \in A \) \( \wedge \) a' \( \notin \) outstanding-refs is-Write \( \exists \) y s \( \wedge \) a'' \( \neq \) a
by auto

show ?thesis using props sb
apply --
apply (rule disjI1)
apply (rule-tac x=sop' in ex)
apply (rule-tac x=a'' in ex)
apply (rule-tac x=v' in ex)
apply (rule-tac x=(x#y)s in ex)
apply (rule-tac x=zs in ex)
apply (simp add: Ghost False)
done
next
assume ?ghst
then obtain y s y z a' L' R' W' where
sb: sb = ys @ Ghost_{ab} A’ L’ R’ W’ # zs and
props: a ∈ A’ a / ∈ outstanding-refs is-Write_{ab} ys
by auto

show ?thesis
using props sb
apply –
apply (rule disjI2)
apply (rule-tac x=A’ in exI)
apply (rule-tac x=L’ in exI)
apply (rule-tac x=R’ in exI)
apply (rule-tac x=W’ in exI)
apply (rule-tac x=(x#ys) in exI)
apply (rule-tac x=zs in exI)
apply (simp add: Ghost{ab} False )
done
qed
qed
qed
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_{ab} volatile a sop v A L R W)
show ?thesis
proof (cases volatile)
case True

from Cons.hyps 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: Write_{ab} 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’]
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_{ab} True domIff)
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

410
by auto

next
case (Ghost, 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)
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 shared-eq R-owns have ∀ a ∈ R. (a ∈ dom S) = (a ∈ dom S')
by (auto simp add: domIff)
from augment-rels-shared-exchange [OF this]
have (augment-rels (dom S') R R) = (augment-rels (dom S) R R).

with Cons.hyps [OF shared-eq' consis']
have release sb (dom (S' ⊕ W R ⊖ A L)) (augment-rels (dom S') R R) =
  release sb (dom (S ⊕ W R ⊖ A L)) (augment-rels (dom S) R R) by simp
then show ?thesis
  by (clarsimp simp add: Ghost domIff)
qed

d Lemma read-only-share-all-shared: \( \forall S. [ a ∈ \text{read-only \{share sb S\}}] \\implies a ∈ \text{read-only S} ∪ \text{all-shared sb} \)
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
  show ?case
  proof (cases x)
    case (Write volatile a sop v A L R W)
    show ?thesis
    proof (cases volatile)
      case True
      with Write 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 Cons show ?thesis by auto
    qed
  next
    case Read with Cons show ?thesis by auto
next
case Prog with Cons show ?thesis by auto
next
case (Ghost, 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

d Lemma read-only-shared-all-until-volatile-write-subset':
\( \forall S. \text{read-only \{share-all-until-volatile-write ts S\} ⊆ \text{read-only S} ∪ (\bigcup ((\lambda(\_,\_,\_,\_,\text{sb},\_,\_,\_,\_). \text{all-shared (takeWhile (Not o is-volatile-Write sb) sb)) ' set ts}))} \)
proof (induct ts)
  case Nil thus ?case by simp
next
  case (Cons t ts)
  obtain p is O R D S sb where
t: t = (p,is,0,sb,D,O,R)
  by (cases t)

  have aargh: (Not o is-volatile-Write sb) = (λa. ¬ is-volatile-Write sb a)
    by (rule ext) auto

  let ?take-sb = (takeWhile (Not o is-volatile-Write sb) sb)
  let ?drop-sb = (dropWhile (Not o is-volatile-Write sb) 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 ∉ (∪ ((λ(·, ·, ·, ·, sb, ·, ·, ·), all-shared (takeWhile (Not o is-volatile-Write sb) sb)) ' set ts))
    have a ∈ all-shared (takeWhile (Not o is-volatile-Write sb) 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:
  (O S, weak-sharing-consistent O sb) ⇒ O ∩ read-only S = {a} ⇒
a ∈ read-only (share sb S) ⇒ a ∈ O ∪ all-acquired sb ⇒ a ∈ all-shared sb

proof (induct sb)
  case Nil thus ?case by auto
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
      owns-ro: O ∩ read-only S = {a} and L-A: L ⊆ A and A-R: A ∩ R = {a} and
      R-owns: R ⊆ O and cons*: weak-sharing-consistent (O ∪ A − R) sb and
        a-share: a ∈ read-only (share sb (S ⊕ W R ⊕ A L)) and
        a-A-all: a ∈ O ∪ A ∪ all-acquired sb
      by (clarsimp simp add: Write True)
from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (\mathcal{S} \oplus_{W} R \oplus_{A} L) = \{\}\)
by (auto simp add: in-read-only-convs)
from Cons.hyps [OF consis' owns-ro' a-share]
show ?thesis
using L-A A-R R-owns owns-ro a-A-all
by (auto simp add: Writeab volatile augment-shared-def restrict-shared-def read-only-def domIff
split: if-split-asm)
next
case False
with Cons Writeab show ?thesis by (auto)
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: \(\mathcal{O} \cap \text{read-only} \mathcal{S} = \{\}\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and
R-owns: \(R \subseteq \mathcal{O}\) and consis': weak-sharing-consistent \((\mathcal{O} \cup A - R)\) sb and
a-share: a \(\in\) read-only (share sb \((\mathcal{S} \oplus_{W} R \oplus_{A} L)\)) and
a-A-all: a \(\in\) \(\mathcal{O} \cup A \cup\) all-acquired sb
by (clarsimp simp add: Ghostab)
from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (\mathcal{S} \oplus_{W} R \oplus_{A} L) = \{\}\)
by (auto simp add: in-read-only-convs)
from Cons.hyps [OF consis' owns-ro' a-share]
show ?thesis
using L-A A-R R-owns owns-ro a-A-all
by (auto simp add: Ghostab augment-shared-def restrict-shared-def read-only-def domIff
split: if-split-asm)
qed

lemma read-only-share-unowned': \(\mathcal{O} S.\)
\([\text{weak-sharing-consistent} \mathcal{O} \text{ sb}; \mathcal{O} \cap \text{read-only} \mathcal{S} = \{\}; \text{ a } \notin \mathcal{O} \cup \text{ all-acquired sb}; \text{ a } \in \text{ read-only} \mathcal{S}]\)
\(\Longrightarrow \) \(a \in\) read-only (share sb \(\mathcal{S}\))
proof (induct sb)
case Nil thus ?case by simp
next
case (Cons x sb)
show ?case
proof (cases x)
case (Writeab volatile a' sop v A L R W)
show ?thesis
proof (cases volatile)
case False
with Cons Writeab show ?thesis by auto
next
case True
from Cons.prems obtain
owns-ro: \(\mathcal{O} \cap \text{read-only} \mathcal{S} = \{\}\) and L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and
R-owns: \(R \subseteq \mathcal{O}\) and consis': weak-sharing-consistent \((\mathcal{O} \cup A - R)\) sb and
a-share: a \(\in\) read-only \(\mathcal{S}\) and
a-notin: a \(\notin\) \(\mathcal{O}\) a \(\notin\) A a \(\notin\) all-acquired sb
by (clarsimp simp add: Writeab True)
from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (\mathcal{S} \oplus_{W} R \oplus_{A} L) = \{\}\)
by (auto simp add: in-read-only-convs)
from a-notin have a-notin': a \(\notin\) \(\mathcal{O}\) \(\mathcal{O}\) \(\mathcal{A}\) \(\mathcal{R}\) \(\mathcal{U}\) all-acquired sb
by auto
from a-share a-notin L-A A-R R-owns have a-ro': a \(\in\) read-only (\(\mathcal{S} \oplus_{W} R \oplus_{A} L\))
by (auto 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: Write sb True)
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
  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 ∉ all-acquired sb
by (clarsimp simp add: Ghost_ab)
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 (auto 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 (clarsimp simp add: Ghost_ab)
qed
qed

lemma release-False-mono:
  ∀ S R. R a = Some False ⇒ outstanding-refs is-volatile-Write_ab sb = {} ⇒
release sb S R a = 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 rels-a: R a = Some False by fact
    then have (augment-rels S R R) a = Some False
    by (auto simp add: augment-rels-def)
    from Cons.hyps [where R = (augment-rels S R 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:
\[ S \forall R. R a = \text{Some False} \implies \text{release (takeWhile (Not o is-volatile-Write\textsubscript{ab}) sb) S R a = Some False} \]

proof (induct sb)
  case Nil thus ?case by simp
next
  case (Cons x sb)
  show ?case
  proof (cases x)
    case (\text{Ghost\textsubscript{ab}} A L R W)
    have rels-a: \( R a = \text{Some False} \) by fact
    then have (augment-rels S R) a = Some False
      by (auto simp add: augment-rels-def)
    from Cons.hyps [where \( R = (\text{augment-rels S R}) \), OF this]
    show ?thesis
      by (clarsimp simp add: Ghost\textsubscript{ab})
  next
    case Write\textsubscript{ab} sb with Cons
    show ?thesis
      by auto
  next
    case Read\textsubscript{ab} sb with Cons
    show ?thesis
      by auto
  next
    case Prog\textsubscript{ab} sb with Cons
    show ?thesis
      by auto
  qed

lemma shared-switch:
\[ S \forall O. \[
\begin{align*}
\text{weak-sharing-consistent O sb; read-only S \cap O = \{\};} \\
S a \neq \text{Some False; share sb S a = Some False}
\end{align*}
\implies a \in O \cup \text{all-acquired sb} \]

proof (induct sb)
  case Nil thus ?case by (auto simp add: read-only-def)
next
  case (Cons x sb)
  have aargh: \( (\text{Not o is-volatile-Write\textsubscript{ab}}) = (\lambda a. \neg \text{is-volatile-Write\textsubscript{ab}} a) \)
    by (rule ext) auto
  show ?case
  proof (cases x)
    case (\text{Ghost\textsubscript{ab}} A L R W)
    from Cons.prems obtain
      dist: \( \text{read-only S \cap O = \{\};} \) and
      share: \( S a \neq \text{Some False} \) and
      share': share sb (\( S \ominus_W R \ominus_A L \)) a = 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 (\( O \cup A \ominus R \)) sb by (clarsimp simp add: \text{Ghost\textsubscript{ab}} aargh)
    from dist L-A A-R R-owns have dist': \( \text{read-only (S \ominus_W R \ominus_A L) \cap (O \cup A \ominus R) = \{\}} \)
      by (auto simp add: in-read-only-consvs)
    show ?thesis
    proof (cases (\( S \ominus_W R \ominus_A L \)) a = Some False)
      case False
      from Cons.hyps [OF consis' dist' this share']
      show ?thesis
        by (auto simp add: \text{Ghost\textsubscript{ab}})
    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 ?thesis
        by (auto simp add: \text{Ghost\textsubscript{ab}})
    qed
next
case (Write\textsubscript{ab} volatile a′ sop v A L R W)
show ?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{ab} True aargh)
from dist L-A A-R R-owns have dist′: read-only \((S \oplus W R \ominus A L)\) \((O \cup A - R)\) = \{\}
by (auto simp add: in-read-only-convs)
show ?thesis
proof (cases \((S \oplus W R \ominus A L)\) a = Some False)
case False from Cons. hyps [OF consis′ dist′ this share′]
show ?thesis by (auto simp add: Write\textsubscript{ab} True)
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 ?thesis by (auto simp add: Write\textsubscript{ab} volatile)
qed
next
case False with Cons show ?thesis by (auto simp add: Write\textsubscript{ab})
qed

next
case Read\textsubscript{ab} with Cons show ?thesis by (auto)
next
case Prog\textsubscript{ab} with Cons show ?thesis by (auto)
qed

lemma shared-switch-release-False:
\(\forall S R. \ [
\text{outstanding-refs is-volatile-Write}_{ab} \ sb = \{\};
\ a \notin \text{dom } S;
\ a \in \text{dom } (\text{share sb } S)]\]
\(\Rightarrow\)
release sb \((\text{dom } S) \ R \ a = \text{Some False}\)

proof (induct sb)
case Nil thus ?case by (auto simp add: read-only-def)
next
case \((\text{Cons } x \ sb)\)
have aargh: \((\text{Not } o \text{-volatile-Write}_{ab}) = (\lambda a. \neg \text{-volatile-Write}_{ab} \ a)\)
by (rule ext) auto
show ?case
proof (cases x)
case \((\text{Ghost}_{ab} A L R W)\)
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′: \text{outstanding-refs is-volatile-Write}_{ab} \ sb = \{\}
by (clarsimp simp add: Ghost\textsubscript{ab} aargh)
show thesis

proof (cases a ∈ R)
case False
  with a-notin have a ∉ dom (S ⊕ W R ⊖ A L)
  by auto
  from Cons.hyps [OF out' this share]
  show thesis
  by (auto simp add: Ghostab)
next
case True
  with a-notin have augment-rels (dom S) R R a = Some False
  by (auto simp add: augment-rels-def split: option.splits)
  from release-False-mono [OF augment-rels (dom S) R R, OF this out']
  show thesis
  by (auto simp add: Ghostab)
qed

next
case Writeab with Cons show thesis by (clarsimp split: if-split-asm)
next
case Readab with Cons show thesis by auto
next
case Progab with Cons show thesis by auto
qed

lemma release-not-unshared-no-write:

∀S R. []
  outstanding-refs is-volatile-Writeab sb = {};
  non-volatile-writes-unshared S sb;
  release sb (dom S) R R a ≠ Some False;
  a ∈ dom (share sb S)]
⇒ a ∉ outstanding-refs is-non-volatile-Writeab sb

proof (induct sb)
case Nil thus thesis case by (auto simp add: read-only-def)
next
case (Cons x sb)
  have aargh: (Not ◦ is-volatile-Writeab) = (λa. ¬ is-volatile-Writeab a)
  by (rule ext) auto
  show thesis
proof (cases x)
case (Ghost, A L R W)
  from Cons.prems obtain
  share: a ∈ dom (share sb (S ⊕ W R ⊖ A L)) and
  rel: release sb
    (dom (S ⊕ W R ⊖ A L)) (augment-rels (dom S) R R) a ≠ Some False and
  out': outstanding-refs is-volatile-Writeab sb = { } and
  nvu: non-volatile-writes-unshared (S ⊕ W R ⊖ A L) sb
  by (clarsimp simp add: Ghostab )

  from Cons.hyps [OF out' nvu rel share]
  show thesis by (auto simp add: Ghostab)
next
case (Write, volatile a' sop v A L R W)
  show thesis
proof (cases volatile)
  case True with Writeab Cons.prems have False by auto
  thus thesis .

417
next
case False
  note not-vol = this
from Cons.prems obtain
  rel: release sb (dom $S$) $\mathcal{R}$ 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$_{ab}$ not-vol)
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 $S$) $\mathcal{R}$ a = Some False.
  with rel have False by simp
  thus ?thesis ..
qed

next
case Read$_{ab}$ with Cons show ?thesis by auto
next
  case Prog$_{ab}$ with Cons show ?thesis by auto
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 $S$) $\mathcal{R}$ 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 \mathcal{O}. \; [\mathcal{O} \cap \text{read-only } S = \{\}; \text{weak-sharing-consistent } \mathcal{O} \text{ sb}; \ a \in \text{read-only } S; \ a \not\in \text{all-shared sb}; \ a \not\in \text{acquired True sb } \mathcal{O} ] 

  \implies \ a \in \text{read-only (share sb } S\text{)}$
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
    note volatile=this
  from Cons.prems
  obtain a-ro: a $\in$ read-only S and a-R: a $\not\in$ R and a-unsh: a $\not\in$ all-shared sb and
  owns-ro: $\mathcal{O} \cap$ read-only S = $\{\}$ and
  L-A: L $\subseteq$ A and A-R: A $\cap$ R = $\{\}$ and R-owns: R $\subseteq$ $\mathcal{O}$ and
consis': weak-sharing-consistent \((\mathcal{O} \cup A - R)\) sb and
\(\text{a-notin}: a \notin \text{acquired True sb} (\mathcal{O} \cup A - R)\)
by (clarsimp simp add: Write\textsubscript{ab}, True)
show ?thesis
proof (cases \(a \in A\))
case True
with \(a\)-R have \(a \in \mathcal{O} \cup A - R\) by auto
from all-shared-acquired-in \([\text{OF this a-unsh}]\)
have \(a \in \text{acquired True sb} (\mathcal{O} \cup A - R)\) by auto
with \(a\)-notin have False by auto
thus ?thesis ..
next
case False
from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (S \oplus W R \ominus A L) = \{\}\)
by (auto simp add: in-read-only-convs)
from a-ro False owns-ro L-A have a-ro': \(a \in \text{read-only} (S \oplus W R \ominus A L)\)
by (auto simp add: in-read-only-convs)
from Cons.hyps [\(\text{OF owns-ro'} consis' a-ro' a-unsh a-notin\)]
show ?thesis
by (clarsimp simp add: Write\textsubscript{ab}, False)
qed
next
case Read\textsubscript{ab} with Cons show ?thesis by (clarsimp)
next
case Prog\textsubscript{ab} with Cons show ?thesis by (clarsimp)
next
case (Ghost\textsubscript{ab}, A L R W)
from Cons.prems
obtain a-ro: \(a \in \text{read-only} S\) and a-R: \(a \notin R\) and a-unsh: \(a \notin \text{all-shared sb and}
owns-ro: \(\mathcal{O} \cap \text{read-only} S = \{\}\) and
L-A: \(L \subseteq A\) and A-R: \(A \cap R = \{\}\) and R-owns: \(R \subseteq \mathcal{O}\) and
consis': weak-sharing-consistent \((\mathcal{O} \cup A - R)\) sb and
\(\text{a-notin}: a \notin \text{acquired True sb} (\mathcal{O} \cup A - R)\)
by (clarsimp simp add: Ghost\textsubscript{ab})
show ?thesis
proof (cases \(a \in A\))
case True
with \(a\)-R have \(a \in \mathcal{O} \cup A - R\) by auto
from all-shared-acquired-in \([\text{OF this a-unsh}]\)
have \(a \in \text{acquired True sb} (\mathcal{O} \cup A - R)\) by auto
with \(a\)-notin have False by auto
thus ?thesis ..
next
case False
from owns-ro A-R R-owns have owns-ro': \((\mathcal{O} \cup A - R) \cap \text{read-only} (S \oplus W R \ominus A L) = \{\}\)
by (auto simp add: in-read-only-convs)
from a-ro False owns-ro R-owns L-A have a-ro': \(a \in \text{read-only} (S \oplus W R \ominus A L)\)
by (auto simp add: in-read-only-convs)
from Cons.hyps [\(\text{OF owns-ro'} consis' a-ro' a-unsh a-notin\)]
show ?thesis
by (clarsimp simp add: Ghost\textsubscript{ab})
qed

419
\textbf{lemma} read-only-share-all-until-volatile-write-unacquired':
\[ \forall S. \text{ownership-distinct } S \& \text{read-only-unowned } S \& \text{weak-sharing-consis } S; \]
\[ \forall i < \text{length } S. (\text{let } (-\cdot, -\cdot, -\cdot, -\cdot, -\cdot, O, R) = S! i \text{ in} \]
\[ a \notin \text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) sb) O \land \]
\[ a \notin \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) sb) \]
\[ a \in \text{read-only } S] \Rightarrow a \in \text{read-only } (\text{share-all-until-volatile-write } ts S) \]
\textbf{proof} (\text{induct } ts)
\textbf{case} Nil thus ?case by simp
\textbf{next}
\textbf{case} (Cons t ts)
\textbf{obtain} p is ORD \theta sb where
\[ t \cdot t = (p, \text{is}, \theta, sb, D, O, R) \]
\textbf{by} (\text{cases } t)
\textbf{have} dist: ownership-distinct (t\#ts) by fact
\textbf{then interpret} ownership-distinct-tl [OF dist]
\textbf{have} dist': ownership-distinct ts.

\textbf{have} aargh: (Not \circ \text{is-volatile-Write}_{ab}) = (\lambda a. \text{is-volatile-Write}_{ab} a)
\textbf{by} (rule ext) auto

\textbf{have} a-ro: a \in \text{read-only } S by fact
\textbf{have} ro-unowned: read-only-unowned S (t\#ts) by fact
\textbf{then interpret} read-only-unowned S t\#ts .
\textbf{have} consis: weak-sharing-consis (t\#ts) by fact
\textbf{then interpret} weak-sharing-consis t\#ts .

\textbf{note} consis' = weak-sharing-consis-tl [OF consis]

\textbf{let} ?take-sb = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) sb)
\textbf{let} ?drop-sb = (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}_{ab}) sb)

\textbf{from} weak-sharing-consis [of 0] t
\textbf{have} consis-sb: weak-sharing-consistent O sb
\textbf{by} force
\textbf{with} weak-sharing-consistent-append [of O ?take-sb ?drop-sb]
\textbf{have} consis-take: weak-sharing-consistent O ?take-sb
\textbf{by} auto

\textbf{have} ro-unowned': read-only-unowned (share ?take-sb S) ts
\textbf{proof}
\textbf{fix} j
\textbf{fix} p j is j i j \theta j sb j
\textbf{assume} j-bound: j < \text{length } ts
\textbf{assume} jth: t\{j\} = (p, \text{is}, i, \theta, sb, D, O, R)
\textbf{show} O\{j\} \cap read-only (share ?take-sb S) = {} \]
\textbf{proof} –
\{ 
\textbf{fix} a
\textbf{assume} a-owns: a \in O\{j\}
\textbf{assume} a-ro: a \in \text{read-only } (share ?take-sb S)
\textbf{have} False
\textbf{proof} –

420
from ownership-distinct [of 0 Suc j] j-bound jth t
have dist: \((O \cup \text{all-acquired sb}) \cap (O_j \cup \text{all-acquired sb}_j) = \{\}\)
  by fastforce

from read-only-unowned [of Suc j] j-bound jth t
have dist-ro: \(O_j \cap \text{read-only } S = \{\}\) by force
show \(\text{?thesis}\)
proof (cases \(a \in (O \cup \text{all-acquired sb})\))
  case True
  with dist a-owns show False by auto
next
  case False
  hence \(a \not\in (O \cup \text{all-acquired } \text{?take-sb})\)
  using all-acquired-append [of \(\text{?take-sb } \text{?drop-sb}\)]
  by auto
  from read-only-share-unowned [OF consis-take this a-ro]
  have \(a \in \text{read-only } S\).
  with dist-ro a-owns show False by auto
qed
qed
qed

from Cons.prems
obtain unacq-ts: \(\forall i < \text{length ts. } (let (\_\_,\_\_,\_\_,\_\_,\_\_) = \text{ts!i in}\)
  a \(\not\in \text{acquired True } \text{(takeWhile } (\text{Not} \circ \text{is-volatile-Write}_\text{sb}) \text{ sb}) \text{ O } \land\)
  a \(\not\in \text{all-shared } \text{(takeWhile } (\text{Not} \circ \text{is-volatile-Write}_\text{sb}) \text{ sb}) \text{ and}\)
  unacq-sb: a \(\not\in \text{acquired True } \text{(takeWhile } (\text{Not} \circ \text{is-volatile-Write}_\text{sb}) \text{ sb}) \text{ O } \land\)
  unsh-sb: a \(\not\in \text{all-shared } \text{(takeWhile } (\text{Not} \circ \text{is-volatile-Write}_\text{sb}) \text{ sb})\)
apply clarsimp
apply (rule that)
apply (auto simp add: t aargh)
done

from read-only-unowned [of 0] t
have owns-ro: \(O \cap \text{read-only } S = \{\}\)
  by force

from read-only-unacquired-share’ [OF owns-ro consis-take a-ro unsh-sb unacq-sb]
have a \(\in \text{read-only } \text{(share } \text{(takeWhile } (\text{Not} \circ \text{is-volatile-Write}_\text{sb}) \text{ sb}) \text{ S}).\)
from Cons.hyps [OF dist’ ro-unowned’ consis’ unacq-ts this]
show \(\text{?case}\)
  by (simp add: t)
qed

lemma not-shared-not-acquired-switch:
\(\forall X \ Y. [a \not\in \text{all-shared sb}; a \not\in X; a \not\in \text{acquired True sb X}; a \not\in Y] \implies a \not\in \text{acquired True sb Y}\)
proof (induct sb)
  case Nil thus \(\text{?case by simp}\)
next
  case (Cons x sb)
  show \(\text{?case}\)
proof (cases x)
case (Write_{\text{ib}} \text{ volatile a' sop v A L R W})
show ?thesis

proof (cases volatile)
case True
from Cons.prems obtain
  a-X: a \notin X \text{ and a-acq: a \notin acquired True sb (X \cup A - R) and}
  a-Y: a \notin Y \text{ and a-R: a \notin R and}
  a-shared: a \notin all-shared sb
  by (clarsimp simp add: Write_{\text{ib}} True)
show ?thesis
proof (cases volatile)
case True
from Cons.prems obtain
  a-X: a \notin X \text{ and a-acq: a \notin acquired True sb (X \cup A - R) and}
  a-Y: a \notin Y \text{ and a-R: a \notin R and}
  a-shared: a \notin all-shared sb
  by (clarsimp simp add: Write_{\text{ib}} True)
show ?thesis

next
case False
with a-X a-Y obtain a-X': a \notin X \cup A - R \text{ 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{ib}} True)
qed

next
case False with Cons.hyps [of X Y] Cons.prems show ?thesis by (auto simp add: Write_{\text{ib}})
qed

next
case Read_{\text{ib}} with Cons.hyps [of X Y] Cons.prems show ?thesis by (auto)
next
case Prog_{\text{ib}} with Cons.hyps [of X Y] Cons.prems show ?thesis by (auto)
next
case (Ghost_{\text{ib}} A L R W)
from Cons.prems obtain
  a-X: a \notin X \text{ and a-acq: a \notin acquired True sb (X \cup A - R) and}
  a-Y: a \notin Y \text{ and a-R: a \notin R and}
  a-shared: a \notin all-shared sb
  by (clarsimp simp add: Ghost_{\text{ib}})
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 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 \text{ 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: Ghost_{\text{ib}})
qed

qed

qed
lemma read-only-share-all-acquired-in':
\[ S = \{ \} ; \text{weak-sharing-consistent } O \text{ sb}; \ a \in \text{read-only (share sb S)} \]
\[ \Rightarrow a \in \text{read-only (share sb Map.empty)} \lor (a \in \text{read-only S} \land a \notin \text{acquired True sb O} \land a \notin \text{all-shared sb} ) \]
proof (induct sb)
case Nil thus ?case by auto
next
case (Cons x sb)
show ?case
proof (cases x)
case (Write\text{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 \text{read-only (share sb (S } \oplus \text{ R } \ominus \text{ A L) and}}
owns-ro: O \cap \text{read-only S} = \{ \} \land
L-A: L \subseteq \text{A and A-R: A } \cap \text{R} = \{ \} \land \text{R-owns: R } \subseteq \text{O and}
consis': weak-sharing-consistent (O \cup \text{A } \ominus \text{ R}) sb
by (clarsimp simp add: Write\text{sb} True)
from owns-ro A-R R-owns have owns-ro': (O \cup \text{A } \ominus \text{ R}) \cap \text{read-only (S } \oplus \text{ R } \ominus \text{ 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 (share sb Map.empty)} \lor
(a \in \text{read-only (S } \oplus \text{ R } \ominus \text{ A L) and a } \notin \text{acquired True sb (O } \cup \text{A } \ominus \text{ R) and a } \notin \text{all-shared sb}).

have a \in \text{read-only (share sb (Map.empty } \oplus \text{ R } \ominus \text{ A L)) \lor}
(a \in \text{read-only S} \land a \notin \text{R} \land a \notin \text{acquired True sb (O } \cup \text{A } \ominus \text{ R) and a } \notin \text{all-shared sb})
proof –
{
assume a-emp: a \in \text{read-only (share sb Map.empty)}
have read-only Map.empty \subseteq \text{read-only (Map.empty } \oplus \text{ R } \ominus \text{ 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 \text{ R } \ominus \text{ A L)).)
} 
moreover
{
assume a-ro: a \in \text{read-only (S } \oplus \text{ R } \ominus \text{ A L) and}

a-not-acq: a \notin \text{acquired True sb (O } \cup \text{A } \ominus \text{ R) and}
a-unsh: a \notin \text{all-shared sb}

have ?thesis
proof (cases a \in \text{read-only S})
case True
with a-ro obtain a-A: a \notin \text{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 \in \text{read-only (Map.empty } \oplus \text{ R } \ominus \text{ A L) by (auto simp add: in-read-only-convs split: if-split-asm)}

have read-only (Map.empty } \oplus \text{ R } \ominus \text{ A L) \subseteq \text{read-only (S } \oplus \text{ R } \ominus \text{ 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-empty a-unsh a-not-acq]
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)
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: \(a \in \text{read-only} (\text{share sb} (S \oplus W R \ominus A L))\) 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 \((O \cup A - R)\) \(\text{sb}\)
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 (clarsimp simp add: in-read-only-convs)

from Cons.hyps [OF owns-ro' consis' a-in]
have hyp: \(a \in \text{read-only} (\text{share sb} \text{Map.empty}) \lor\)
  \((a \in \text{read-only} (S \oplus W R \ominus A L) \land a \notin \text{acquired} \text{ True sb} (O \cup A - R) \land a \notin \text{all-shared sb})\).

have \(a \in \text{read-only} (\text{share sb} (\text{Map.empty} \oplus W R \ominus A L)) \lor\)
  \((a \in \text{read-only} S \land a \notin R \land a \notin \text{acquired True sb} (O \cup A - R) \land a \notin \text{all-shared sb})\)
proof –
{assume a-emp: \(a \in \text{read-only} (\text{share sb Map.empty})\)
have read-only Map.empty \(\subseteq \) 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 \(a \in \text{read-only} (\text{share sb} (\text{Map.empty} \oplus W R \ominus A L)).\)
}
moreover
{assume a-ro: \(a \in \text{read-only} (S \oplus W R \ominus A L)\) and
  a-not-acq: \(a \notin \text{acquired True sb} (O \cup A - R)\) and
  a-unsh: \(a \notin \text{all-shared sb}\)
have ?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)

424
with True a-not-acq a-unsh R-owns owns-ro

show \text{thesis}
  by auto

next
case False

with a-ro have a-ro-empty: a \in read-only (\text{Map.empty} \oplus_{\text{W}} R \ominus_{\text{A}} L)
  by (auto simp add: in-read-only-convs split: if-split-asm)

have read-only (\text{Map.empty} \oplus_{\text{W}} R \ominus_{\text{A}} L) \subseteq \text{read-only} (S \oplus_{\text{W}} R \ominus_{\text{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_{\text{W}} R \ominus_{\text{A}} L) = \{}
  by blast

from \text{read-only-unacquired-share'} [OF owns-ro-empty consis' a-ro-empty a-unsh a-not-acq]

have a \in \text{read-only} (\text{share sb} (\text{Map.empty} \oplus_{\text{W}} R \ominus_{\text{A}} L)).

thus \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-consistent S ts
assumes ro-unowned: read-only-unowned S ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, sb, D, O, R)

assumes a-unacquired-others: \forall j < length ts. i \neq j \rightarrow
  (let (\_\_\_, \_\_\_, \_\_\_, sb_j) = ts!j in
    a \notin \text{acquired} True (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb_j}) sb_j) O \land
    a \notin \text{all-shared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb_j}) sb_j))
assumes a-ro-share: a \in \text{read-only} (\text{share sb S})
s shows a \in \text{read-only} (\text{share} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb)
  (\text{share-all-until-volatile-write ts S}))

proof

from consis
interpret sharing-consistent 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 \text{read-only} S = {\}.
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} sb.
moreover

let ?take-sb = (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb)
let ?drop-sb = (\text{dropWhile} (\text{Not} \circ \text{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 ∈ read-only (share sb Map.empty)
with share-append [of ?take-sb ?drop-sb]
  have a-in': a ∈ read-only (share ?drop-sb (share ?take-sb Map.empty))
  by auto

  have owns-empty: O ∩ read-only Map.empty = {}
  by auto

  from weak-sharing-consistent-preserves-distinct [OF weak-consis-take owns-empty]
  have acquired True ?take-sb O ∩ read-only (share ?take-sb Map.empty) = {}

  from read-only-share-all-acquired-in [OF this weak-consis' a-in']
  have a ∈ read-only (share ?drop-sb Map.empty) ∨ a ∈ read-only (share ?take-sb Map.empty) ∧ a /∈
  all-acquired ?drop-sb.

  moreover

  { assume a-ro-drop: a ∈ read-only (share ?drop-sb Map.empty)
    have read-only Map.empty ⊆ 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 ∈ read-only (share ?take-sb Map.empty)
    assume a-unacq-drop: a /∈ all-acquired ?drop-sb
    from read-only-share-unowned-in [OF weak-consis-take a-ro-take]
    have a ∈ O ∪ all-acquired ?take-sb by auto
    hence a ∈ O ∪ 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 /∈ acquired True sb O
  assume a-unsh: a /∈ all-shared sb
  with all-shared-append [of ?take-sb ?drop-sb]
  obtain a-notin-take: a /∈ all-shared ?take-sb and a-notin-drop: a /∈ all-shared ?drop-sb
  by auto
  have thesis
  proof (cases a ∈ acquired True ?take-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 ..
    next
    case False

426
with a-unacquired-others i-bound ts-i a-notin-take
have a-unacq': ∀ j < length ts.
(let (-,-,sbj,-,O,,-) = ts[j] in
  a /∈ acquired True (takeWhile (Not o is-volatile-Write sbj) O ∧
  a /∈ all-shared (takeWhile (Not o is-volatile-Write sbj ) 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]
  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. i̸=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:
  ∀ R S S. ∀ a ∈ all-shared sb. (a ∈ S') = (a ∈ S) ==> release sb S' R = 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\textsubscript{sb} volatile a' sop v A L R W)
  show ?thesis
proof (cases volatile)
case True
  note volatile=\textit{this}
  from Cons.hyps \{of (S' \cup R - L) (S \cup R - L) Map.empty\} Cons.prems
  show ?thesis
  by (auto simp add: Write\textsubscript{sb} volatile)
next
case False with Cons Write\textsubscript{sb} show ?thesis by auto
qed
next
case Read\textsubscript{sb} with Cons show ?thesis by auto
next
case Prog\textsubscript{sb} with Cons show ?thesis by auto
next
case (Ghost\textsubscript{sb} A L R W)
  from augment-rels-shared-exchange \{of R S S' R\} Cons.prems
  have augment-rels S'R R = augment-rels S R R
  by (auto simp add: Ghost\textsubscript{sb})
with Cons.hyps \{of (S' \cup R - L) (S \cup R - L) augment-rels S R R\} Cons.prems
  show ?thesis
  by (auto simp add: Ghost\textsubscript{sb})
qed
qed
lemma release-append-Prog\textsubscript{sb}:
\forall S R. (release (takeWhile (Not ◦ is-volatile-Write\textsubscript{sb}) \{sb \at \{Prog\textsubscript{sb} p_1 p_2 mis\}\}) S R) =
(release (takeWhile (Not ◦ is-volatile-Write\textsubscript{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\textsubscript{sb}, m\textsubscript{sb}, S\textsubscript{sb}) \Rightarrow_s\textsubscript{sb} (ts\textsubscript{sb}', m\textsubscript{sb}', S\textsubscript{sb}')
assumes valid-own: valid-ownership S\textsubscript{sb} ts\textsubscript{sb}
assumes valid-sb-reads: valid-reads m\textsubscript{sb} ts\textsubscript{sb}
assumes valid-hist: valid-history program-step ts\textsubscript{sb}
assumes valid-sharing: valid-sharing S\textsubscript{sb} ts\textsubscript{sb}
assumes tmprs-distinct: tmprs-distinct ts\textsubscript{sb}
assumes valid-sops: valid-sops ts\textsubscript{sb}
assumes valid-dd: valid-data-dependency ts\textsubscript{sb}
assumes load-temps-fresh: load-temps-fresh ts\textsubscript{sb}
assumes enough-flushs: enough-flushs ts\textsubscript{sb}
assumes valid-program-history: valid-program-history ts\textsubscript{sb}
assumes valid: valid ts\textsubscript{sb}
assumes sim: (ts\textsubscript{sb}, m\textsubscript{sb}, S\textsubscript{sb}) \sim (ts, m, S)
assumes safe-reach: safe-reach-direct safe-delayed (ts, m, S)
shows valid-ownership S\textsubscript{sb}' ts\textsubscript{sb}' \land valid-reads m\textsubscript{sb}' ts\textsubscript{sb}' \land valid-history program-step ts\textsubscript{sb}' /\land
valid-sharing S\textsubscript{sb}' ts\textsubscript{sb}' \land tmprs-distinct ts\textsubscript{sb}' \land valid-data-dependency ts\textsubscript{sb}' \land
valid-sops ts\textsubscript{sb}' \land load-temps-fresh ts\textsubscript{sb}' \land enough-flushs ts\textsubscript{sb}' /\land
valid-program-history \(ts_{sb}' \land valid \ ts_{sb}' \land
(\exists ts', S', m', (ts, m, S) \Rightarrow^d (ts', m', S') \land
(ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S'))\)

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 sbh. sbh .
interpret valid-ownership \(S_{sb} ts_{sb}\) by fact
interpret valid-reads \(m_{sb} ts_{sb}\) by fact
interpret valid-sharing \(S_{sb} ts_{sb}\) by fact
interpret tmsgs-distinct \(ts_{sb}\) 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-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 ?thesis
proof (cases)
case (Memop i \(p_{sb}\) \(is_{sb}\) \(\theta_{sb}\) sb \(D_{sb}\) \(O_{sb}\) \(R_{sb}\) \(is_{sb}' \land \theta_{sb}' \land \theta_{sb}' \land D_{sb}' \land O_{sb}' \land R_{sb}'\))
then obtain
\(ts_{sb}' \land ts_{sb}' = ts_{sb}[i := (\(p_{sb}\), \(is_{sb}'\), \(\theta_{sb}'\), \(sb'\), \(D_{sb}'\), \(O_{sb}'\), \(R_{sb}'\))]\) and
i-bound: \(i < \) length \(ts_{sb}\) and
\(ts_{sb}\)-i: \(ts_{sb} ! i = (p_{sb}, is_{sb}, \theta_{sb}, sb, D_{sb}, O_{sb}, R_{sb})\) and
sbh-step: \((is_{sb}', \theta_{sb}', sb', m_{sb}, D_{sb}', O_{sb}', R_{sb}', S_{sb}') \rightarrow_{sbh}\)
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: \(\forall i < \) length \(ts_{sb}\) and
\(ts\)-sim: \(\forall i < \) length \(ts_{sb}\).
let \((p, is_{sb}, \theta, sb, D_{sb}, O_{sb}, R) = ts_{sb} ! i\);
  suspends = dropWhile (Not \circ is\-volatile\-Write_{sb}) sb
in \(\exists i\ D.\ instrs\ suspends \odot is_{sb} = is \odot\\ prog\-instrs\ suspends\) and

429
\[ \mathcal{D}_{sb} = (\mathcal{D} \lor \text{outstanding-refs \text{-} volatile-Write}_{sb} \neq \{} \} ) \land \\
\text{ts} ! i = \\
(\text{hd-prog } p \text{ suspends,} \\
is, \\
\emptyset \mid (\text{dom } \emptyset - \text{read-tmps suspends}), () , \\
\mathcal{D} , \\
\text{acquired True (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) } \mathcal{O}_{sb}, \\
\text{release (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) } \mathcal{O}_{sb}) (\text{dom } \mathcal{S}_{sb}) \mathcal{R}_{sb}) \\
\text{by cases blast} \]

\text{from } \text{i-bound \ leq } \text{have } \text{i-bound'}: i < \text{length ts} \\
\text{by auto} \\

\text{have } \text{split-sb}: sb = \text{takeWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb } @ \text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb} \\
(is sb = ?\text{take-sb@?drop-sb}) \\
\text{by simp} \\

\text{from } \text{ts-sim [rule-format, OF i-bound] ts}_{sb} - i \text{ obtain suspends is } \mathcal{D} \text{ where} \\
suspends: suspends = \text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb} \text{) sb} \text{ and} \\
is-sim: \text{instrs suspends } @ is_{sb} = is \text{ @ prog-instrs suspends and} \\
\mathcal{D}: \mathcal{D}_{sb} = (\mathcal{D} \lor \text{outstanding-refs \text{-} volatile-Write}_{sb} \neq \{} \} ) \land \\
ts-i: \text{ts} ! i = \\
(\text{hd-prog } p_{sb} \text{ suspends,} \\
is_{sb} \mid (\text{dom } \emptyset_{sb} - \text{read-tmps suspends}), () , \mathcal{D} , \text{acquired True } ?\text{take-sb } \mathcal{O}_{sb}, \\
\text{release } ?\text{take-sb} (\text{dom } \mathcal{S}_{sb}) \mathcal{R}_{sb}) \\
\text{by (auto simp add: Let-def)} \\

\text{from } \text{sbh-step-preserves-valid [OF i-bound ts}_{sb} - i \text{ sbh-step valid]} \\
\text{have valid'}: \text{valid ts} _{sb}' \\
\text{by (simp add: ts}_{sb}') \\

\text{from } \mathcal{D} \text{ have } \mathcal{D}_{sb} : \mathcal{D}_{sb} = (\mathcal{D} \lor \text{outstanding-refs \text{-} volatile-Write}_{sb} ?\text{drop-sb} \neq \{} \}) \\
\text{apply} - \\
\text{apply (case-tac outstanding-refs \text{-} volatile-Write}_{sb} \text{ sb } = \{ \}) \\
\text{apply (fastforce simp add: outstanding-refs-conv dest: set-dropWhileD)} \\
\text{apply (clarsimp simp add: outstanding-refs-non-empty-dropWhile)} \\
\text{apply blast} \\
\text{done} \\

\text{let } ?ts' = ts[i := (p_{sb}, is_{sb}, \emptyset_{sb}, (), \mathcal{D}_{sb}, \text{acquired True sb } \mathcal{O}_{sb}, \\
\text{release sb } (\text{dom } \mathcal{S}_{sb}) \mathcal{R}_{sb})] \\
\text{have i-bound-ts'}: i < \text{length } ?ts' \\
\text{using i-bound'} \\
\text{by auto} \\
\text{hence ts'} - i: ?ts'_{i} = (p_{sb}, is_{sb}, \emptyset_{sb}, () , \\
\mathcal{D}_{sb} , \text{acquired True sb } \mathcal{O}_{sb}, \text{release sb } (\text{dom } \mathcal{S}_{sb}) \mathcal{R}_{sb}) \\
\text{by simp} 

430
from local.sharing-consis-axioms
have sharing-consis-ts sb: sharing-consistent $S_{sb}$ $ts_{sb}$.
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 $\emptyset_{sb}$ (hd-prog $p_{sb}$ (?take-sb@?drop-sb))
(?take-sb@?drop-sb)
by (simp)

have last-prog-hd-prog: last-prog (hd-prog $p_{sb}$ sb) ?take-sb = (hd-prog $p_{sb}$ ?drop-sb)
proof -
from last-prog-hd-prog-append' [OF h-consis] last-prog
have last-prog (hd-prog $p_{sb}$ ?drop-sb) ?take-sb = hd-prog $p_{sb}$ ?drop-sb
by (simp)
moreover
have last-prog (hd-prog $p_{sb}$ (?take-sb @ ?drop-sb)) ?take-sb =
ultimately show \(?\text{thesis}\)
\textbf{by} (simp)

\textbf{qed}

\textbf{from} valid-write-sops [OF i-bound ts\textsubscript{sb}-i]
\textbf{have} \(\forall\text{sop}\in\text{write-sops} \ (\text{?take-sb}@\text{?drop-sb}). \ \text{valid-sop} \text{sop}\)
\textbf{by} (simp)

\textbf{then obtain} valid-sops-take: \(\forall\text{sop}\in\text{write-sops} \ (\text{?take-sb}). \ \text{valid-sop} \text{sop}\)
valid-sops-drop: \(\forall\text{sop}\in\text{write-sops} \ (\text{?drop-sb}). \ \text{valid-sop} \text{sop}\)
\textbf{apply} (simp only: write-sops-append)
\textbf{apply} auto
\textbf{done}

\textbf{from} read-tmps-distinct [OF i-bound ts\textsubscript{sb}-i]
\textbf{have} distinct-read-tmps (\(?\text{take-sb}@\text{?drop-sb}\) \text{read-tmps-take-drop}: \text{read-tmps} ?take-sb \cap \text{read-tmps} ?drop-sb = \{\} and
\text{distinct-read-tmps-drop: distinct-read-tmps} ?drop-sb
\textbf{by} (simp only: distinct-read-tmps-append)

\textbf{from} history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]
\textbf{have} \(\text{hist-consis}'\): history-consistent \(\emptyset\text{sb} \ (\text{hd-prog} \ p\text{sb} \ ?\text{drop-sb}) \ ?\text{drop-sb}\)
\textbf{by} (simp add: last-prog-hd-prog)

\textbf{proof —}
\textbf{from} release-append [of \(?\text{take-sb} \ ?\text{drop-sb}\)]
\textbf{have} release sb (dom \(S_{\text{sb}}\)) \(R_{\text{sb}}\) =
release \(\text{dom} \emptyset\text{sb}\) (dom \(S_{\text{sb}}\)) \(R_{\text{sb}}\)
\textbf{by} simp
\textbf{also}
\textbf{have} dist: ownership-distinct ts\textsubscript{sb} by fact
\textbf{have} consis: sharing-consis \(S_{\text{sb}}\) ts\textsubscript{sb} by fact

\textbf{have} release \(\text{dom} \emptyset\text{sb}\) (dom \(\text{share} \emptyset\text{sb} \ S_{\text{sb}}\)) (release \(?\text{take-sb} \ ?\text{drop-sb}\) (dom \(S_{\text{sb}}\)) \(R_{\text{sb}}\)) =

release \(\text{dom} \emptyset\text{sb}\) (dom \(S_{\text{sb}}\)) (release ?take-sb (dom \(S_{\text{sb}}\)) \(R_{\text{sb}}\))
\textbf{apply} (simp only: \(S\))
\textbf{apply} (rule release-shared-exchange-weak [rule-format, OF - weak-consis-drop])
\textbf{apply} (rule share-all-until-volatile-write-thread-local [OF dist consis i-bound ts\textsubscript{sb}-i, symmetric])
\textbf{using} acquired-all-acquired [of True ?take-sb \(O_{\text{sb}}\)] all-acquired-append [of ?take-sb ?drop-sb]
\textbf{by} auto
\textbf{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, ?sb flush ?drop-sb m, D sb,acquired True sb O sb, share ?drop-sb S) \checkmark.

from acquired-takeWhile-non-volatile-Write sb
have acquired-take-sb: acquired True ?take-sb O sb ⊆ O sb ∪ all-acquired ?take-sb .

from sbh-step
show ?thesis
proof (cases)
case (SBHReadBuffered a v volatile t)
then obtain
is sb: is sb = Read volatile a t # is sb’ and
O sb’: O sb’=O sb and
D sb’: D sb’=D sb and
θ sb’: θ sb’= θ sb(t→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

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

433
from clean-no-outstanding-volatile-Write \_{sb} \ [\text{OF } i\text{-bound } ts_{sb}\text{-i}] \ volatile-clean
have volatile-cond: volatile \rightarrow\ outstanding-refs \ is-volatile-Write \_{sb} \ sb = \{\}
by auto

from buffered-val-witness \ [\text{OF } buf\text{-v}] \ obtain \ volatile' \ sop' A' L' R' W'
where
witness: Write \_{sb} \ volatile' \ a \ sop' v \ A' L' R' W' \in \ set \ sb
by auto

{\}
fix j p\_j is_{sbj} \ O\_j R\_j D_{sbj} \ \emptyset_{sbj} \ sb\_j
assume j-bound: j < \text{length } ts_{sb}
assume neq-i-j: i \neq j
assume jth: ts_{sb}\_j = (p\_j, is_{sbj}, \ \emptyset_{sbj}, \ sb\_j, D_{sbj}, O\_j, R\_j)
assume non-vol: \neg \text{volatile}
have a \notin O\_j \cup \text{all-acquired } sb\_j
proof
assume a-j: a \in O\_j \cup \text{all-acquired } sb\_j
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)

from ts-sim \ [\text{rule-format, OF } j\text{-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 \ @ \ \text{prog-instrs } suspends\_j \ and
D\_j: D_{sbj} = (D\_j \lor \text{outstanding-refs } \text{is-volatile-Write } \_sb \ sb\_j \neq \{\}) \ and
ts\_j: ts\_j = (\hd-prog p\_j \ suspends\_j, is\_j,
\ \emptyset_{sbj} \mid (\text{dom } \emptyset_{sbj} - \text{read-tmps } suspends\_j).(),
D\_j, \text{acquired True } ?take-sb\_j O\_j, \text{release } ?take-sb\_j (\text{dom } S_{sb}) R\_j)
by (auto simp add: Let-def)

from a-j ownership-distinct \ [\text{OF } i\text{-bound } j\text{-bound } neq-i-j \ ts_{sb}\text{-i } j\text{th}]\nhave a-notin-sb: a \notin O_{sb} \cup \text{all-acquired } sb
by auto
with acquired-all-acquired \ [\text{of } True \ sb \ O_{sb}]\nhave a-not-acq: a \notin \text{acquired True } sb \ O_{sb} \ by blast
with access-cond’ non-vol
have a-ro: a \in \text{read-only } (\text{share } ?drop-sb \ S)
by auto
from read-only-share-unowned-in \ [\text{OF } \text{weak-consis-drop } a\text{-ro}] \ a\text{-notin-sb}
aquired-all-acquired \ [\text{of } True \ ?take-sb \ O_{sb}]
all-acquired-append \ [\text{of } ?take-sb \ ?drop-sb]
have a-ro-shared: a \in \text{read-only } S
by auto
from reals-nv-cond \ [\text{rule-format, OF } \text{non-vol } j\text{-bound } [\text{simplified leq}] \ neq-i-j] \ ts\_j

434
have \( a \notin \text{dom} \ (\text{release} \ ?\text{take-sb}_j \ (\text{dom} \ (S_{sb})) \ \mathcal{R}_j) \)
by auto
with \( \text{dom-release-takeWhile} \ [\text{of} \ sb_j \ (\text{dom} \ (S_{sb})) \ \mathcal{R}_j] \)

obtain
a-rels\( _j \): \( a \notin \text{dom} \ \mathcal{R}_j \) and
a-shared\( _j \): \( a \notin \text{all-shared} \ ?\text{take-sb}_j \)
by auto

have \( a \notin \bigcup (\lambda (\cdot, \cdot, \cdot, sb, \cdot, \cdot, \cdot). \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb)) \)

set \( ts_{sb} \)

proof ~
{
fix \( k \ p \ k \ i \ k \ sb \ k \ D_k \ \mathcal{O}_k \ \mathcal{R}_k \)
assume k-bound: \( k < \text{length} \ ts_{sb} \)
assume ts-k: \( ts_{sb} ! k = (p_k, i_k, D_k, \mathcal{O}_k, \mathcal{R}_k) \)
assume a-in: \( a \in \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_k) \)
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 \[\text{OF j-bound k-bound False} \ [\text{symmetric}] \ jth \ ts-k \] a-j
have \( a \notin (\mathcal{O}_k \cup \text{all-acquired} \ sb_k) \) by auto
with all-shared-acquired-or-owned \[\text{OF} \ \text{sharing-consis} \ [\text{OF k-bound ts-k}] \] a-in
show False
using all-acquired-append \[\text{OF} \ \text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_k \]
dropWhile \( \text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_k \)
all-shared-append \[\text{OF} \ \text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_k \]
dropWhile \( \text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_k \) by auto
qed

thus \?thesis by (fastforce simp add: in-set-conv-nth)

qed

with \( \text{a-ro-shared} \)

read-only-shared-all-until-volatile-write-subset' \[\text{OF} \ ts_{sb} S_{sb} \]

have \( \text{a-ro-shared}_{sb} \): \( a \in \text{read-only} \ S_{sb} \)
by (auto simp add: S)

with \( \text{read-only-unowned} \ [\text{OF} \ \text{j-bound jth}] \)

have \( \text{a-notin-owns-j} \): \( a \notin \mathcal{O}_j \)
by auto

have own-dist: ownership-distinct \( ts_{sb} \) by fact
have share-consis: sharing-consis \( S_{sb} \) \( ts_{sb} \) by fact
from sharing-consistent-share-all-until-volatile-write \[\text{OF} \ \text{own-dist share-consis i-bound ts}_{sb-i} \]

have consis': sharing-consistent \( S \) \( \text{acquired} \ True \ ?\text{take-sb} \ \mathcal{O}_{sb} \) \?drop-sb
by (simp add: \( S \))
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 \( j \) \( S_{sb} \))
by (auto simp add: domIff \( S \) 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 \( O_j \) ?take-sb ?drop-sb]
have weak-consis-drop:weak-sharing-consistent \( O_j \) ?take-sb
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
have a \( \notin \) all-acquired ?take-sb
by auto
with a-j a-notin-owns-j
have a-drop: \( a \in \text{all-acquired} \) ?drop-sb
using all-acquired-append [of ?take-sb ?drop-sb]
by simp

from i-bound j-bound leq j-bound-ts': \( j < \text{length} \) ?ts'
by auto

note conflict-drop = a-drop [simplified suspends \( j \) [symmetric]]
from split-all-acquired-in [OF conflict-drop]
show False
proof
  assume \( \exists \) sop a' v ys 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; 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-bound jth]
have sharing-consistent \( S_{sb} \) \( O_j \) \( sb_j \).
then have A'-R': A' \( \cap \) R' = \{\}
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 jth]
have causal-program-history is\( sb_j \) \( sb_j \).
then have cph: 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\( j \))

436
from ts\_j neq-i-j j-bound
have ts\_j': ?ts\_j = (hd-prog p\_j suspends\_j, is\_j,
\ Dome \ theta_{sbj} \ (\ \theta_{sbj} \ \theta_{sbj} - \ \text{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 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 sb\_j.
from reads-consistent-flush-all-until-volatile-write [OF \ \text{valid-ownership-and-sharing} S\_sb\_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\_i jth]
have outstanding-refs is-Write\_sb \ ?drop-sb \ \cap \ \text{outstanding-refs is-non-volatile-Read\_sb}
suspends\_j = \{\}
by (simp add: suspends\_j)
from reads-consistent-flush-independent [OF this \ \text{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@[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 \ \text{sop} \ \in \ \text{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 \ \text{sop} \ \in \ \text{write-sops (?take-sb\_j. valid-sop sop and}
valid-sops-drop: \ \forall \ \text{sop} \ \in \ \text{write-sops (ys@[Write\_sb True a' sop' v' A'L'R' W'])} \ \ \text{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\_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 jth]

have h-consis:

history-consistent \touchsbj (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 \touchsbj (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@[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_j'=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

438
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** 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
( 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$_j'$,
  $\emptyset_{sbj}$ |' (dom $\emptyset_{sbj}$ = read-tmps $zs$),
  (,), True, acquired True $ys$ (acquired True ?take-sb $O_j$) $\cup$ $A'$ $- R'$,$R_j'$],
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' **have** i-bound-ys: $i < \text{length } ?ts-ys$
**by** auto

**from** i-bound' neq-i-j
**have** ts-ys-i: ?ts-ys!$i$ = (p$_{sb}$, is$_{sb}$, $\emptyset_{sb}$, ()),
  $D_{sb}$, 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 $\in$ 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
**with** a-A'
**show** False
  **by** (simp add: share-append in-read-only-convs)
**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** A' L' R' W' $ys$ $zs$ **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

439
from valid-program-history [OF j-bound jth]
have causal-program-history is_{sbj} sb_j,

then have cpl: 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_j’ = (hd-prog p_j suspends_j, is_j,
\theta_{sbj} | (dom \theta_{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 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_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} sb_j.
from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing
S_{sb} 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_i} jth]
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_j O_j)
(\text{flush} ?drop-sb m) suspends_j.

hence reads-consis-ys: reads-consistent True (acquired True ?take-sb_j O_j)
(\text{flush} ?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 jth]
have \forall sop\in\text{write-sops} (?take-sb_j \cap ?suspends). valid-sop sop
by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain valid-sops-take: \forall sop\in\text{write-sops} (?take-sb_j). valid-sop sop and
valid-sops-drop: \forall sop\in\text{write-sops} (ys@[Ghost_{sb} A’ L’ R’ W’]). valid-sop sop

440
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound jth]
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
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 \sb \j (hd-prog p_j (?take-sb @ suspends_j)) (?take-sb @ 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 \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-j]
have no-vol-read: outstanding-refs is-volatile-Read_{sb} \j
  (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'])

441
(acquired True ?take-sb \(O_j\)) =
(acquired True ys (acquired True ?take-sb \(O_j\)) \(\cup\) \(A' - R'\)
by (simp add: acquired-append)

\textbf{from} \flush-store-buffer-append \textbf{where} \(sb = ys @{\text{\textit{Ghost}} sb A' L' R' W'}\) \textbf{and} \(sb' = zs\), simplified,
OF \(j\)-bound-ts' is\(j\) [simplified split-suspends] cph [simplified suspends]
ts' \textbf{where} \(sb = ys \cup A' - R'\)
miş-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], \textbf{where} \(S = \text{\textit{share}} (?drop-sb \(S\))\)
\textbf{obtain} \(is_j' R_j'\) \textbf{where}
is\(j\)'s instrs zs @ is\(sbj\) = is\(j\)' @ prog-instrs zs \textbf{and}
steps-ys: (?ts', flush ?drop-sb m, share ?drop-sb \(S\)) \(\Rightarrow_d^*\)
(\(?ts'\)[j] := (last-prog
(hd-prog p_j (\text{\textit{Ghost}} sb A' L' R' W'# zs)) (ys@[\text{\textit{Ghost}} sb A' L' R' W']),
is\(j\)',
\(\emptyset_{sbj} \mid i\) (dom \(\emptyset_{sbj} - \text{read-tmps} zs),
(),
\textbf{dest} \(D_j \lor \text{outstanding-refs} \text{\textit{is-volatile-Write}}_{sb} (ys@[\text{\textit{Ghost}} sb A' L' R' W']) \neq \{\},
acquired True ys (acquired True ?take-sb \(O_j\)) \(\cup\) \(A' - R', R_j'\)],
flush (ys@[\text{\textit{Ghost}} sb A' L' R' W']) (flush ?drop-sb m),
share (ys@[\text{\textit{Ghost}} sb A' L' R' W']) (share ?drop-sb \(S\))
(\(is (\text{\textit{\textbullet \textbullet \textbullet}}) \Rightarrow_d^*\) (\(?ts-ys, ?m-ys, ?\text{\textit{shared-ys}}\))
by (auto simp add: acquired-append)

\textbf{from} i-bound' \textbf{have} i-bound-ys: i < length ?ts-ys
by auto

\textbf{from} i-bound' \textbf{neq-i-j}
\textbf{have} ts-ys-i: (?ts-ys!i = (p_{sb}, is_{sb}, \emptyset_{sb}, ()),
\textbf{dest} \(D_{sb} \), acquired True sb \(O_{sb}\), release sb (dom \(S_{sb}\)) \(R_{sb}\)
by simp
\textbf{note} conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

\textbf{from} safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
\textbf{have} safe-delayed (?ts-ys, ?m-ys, ?\text{\textit{shared-ys}}).

\textbf{from} safe-delayedE [OF this i-bound' ts-ys-i, simplified is\(sb\)] \textbf{non-vol a-not-acq}
\textbf{have} a ∈ read-only (share (ys@[\text{\textit{Ghost}} sb A' L' R' W']) (share ?drop-sb \(S\))
apply cases
apply (auto simp add: Let-def is\(sb\))
done

\textbf{with} a-A'
\textbf{show} False
by (simp add: share-append in-read-only-convs)
qed
qed
\begin{align*}
\text{assume a-in: } & a \in \text{read-only (share dropWhile (Not o is-volatile-Write}_{sb} sb) S)} \\
\text{assume nv: } & \neg \text{volatile} \\
\text{have a \in read-only (share sb } S_{sb}) \\
\text{proof (cases a \in O}_{sb} \cup \text{all-acquired sb)} \\
\text{case True} \\
& \text{from share-all-until-volatile-write-thread-local’ [OF ownership-distinct-ts}_{sb} sharing-consis-ts}_{sb} \text{i-bound ts}_{sb-i} \text{True a-in} \\
& \text{show ?thesis} \\
& \text{by (simp add: S read-only-def)} \\
\text{next} \\
\text{case False} \\
& \text{from read-only-share-unowned [OF weak-consis-drop - a-in] False acquired-all-acquired [of True ?take-sb } O_{sb}] \text{all-acquired-append [of ?take-sb ?drop-sb]} \\
& \text{have a-ro-shared: } a \in \text{read-only S} \\
& \text{by auto} \\
& \text{have a } \notin \bigcup (\lambda (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot). \\
& \text{all-shared (takeWhile (Not o is-volatile-Write}_{sb} sb)) i \text{ set ts}_{sb}) \\
\text{proof -} \\
& \{ \\
& \text{fix k p_k is_k d_k o_k r_k} \\
& \text{assume k-bound: } k < \text{length ts}_{sb} \\
& \text{assume ts-k: } t_{sb} ! k = (p_k, is_k, d_k, o_k, r_k) \\
& \text{assume a-in: } a \in \text{all-shared (takeWhile (Not o is-volatile-Write}_{sb} sb)_k} \\
& \text{have False} \\
& \text{proof (cases k=i)} \\
& \text{case True with False ts}_{sb-i} \text{ ts-k a-in} \\
& \text{all-shared-acquired-or-owned [OF sharing-consis [OF k-bound ts-k]]} \\
& \text{all-shared-append [of takeWhile (Not o is-volatile-Write}_{sb} sb)_k} \\
& \text{dropWhile (Not o is-volatile-Write}_{sb} sb)_k] \text{ show False by auto} \\
\text{next} \\
\text{case False} \\
& \text{from rels-nv-cond [rule-format, OF nv k-bound [simplified leq] False [symmetric]} \\
& \text{ts-sim [rule-format, OF k-bound] ts-k} \\
& \text{have a } \notin \text{ dom (release (takeWhile (Not o is-volatile-Write}_{sb} sb)_k) (dom (S}_{sb}) \\
& \text{R}_k) \\
& \text{by (auto simp add: Let-def)} \\
& \text{with dom-release-takeWhile [of sb}_k \text{ (dom (S}_{sb}) \text{ R}_k] \\
& \text{obtain} \\
& \text{a-rels_j: } a \notin \text{ dom } R_k \text{ and} \\
& \text{a-shared_j: } a \notin \text{ all-shared (takeWhile (Not o is-volatile-Write}_{sb} sb)_k} \\
& \text{by auto} \\
& \text{with False a-in show ?thesis} \\
& \text{by auto} \\
\end{align*}

443
thus ?thesis by (fastforce simp add: in-set-conv-nth)

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 .

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)
case False
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.
then
have non-volatile-owned-or-read-only False S sb O sb (sb@[Read sb volatile a t v])
  using access-cond' False non-vol-ro-reduction
  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 O sb' S sb' )
next
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 O sb sb.
then
have non-volatile-owned-or-read-only False S sb O sb (sb@[Read sb True a t v])
  using True
  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 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]) ⊆
    outstanding-refs is-volatile-Write sb
    by (auto simp add: outstanding-refs-append)
  have all-acquired (sb@[Read sb volatile a t v]) ⊆ 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}) (sb @ [Read_{sb} volatile a t v]))) $O_{sb}$
    (dropWhile (Not ◦ is-volatile-Write_{sb}) (sb @ [Read_{sb} volatile a t v])) \subseteq \text{read-only-reads (acquired True (takeWhile (Not ◦ 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 $O_{sb} \cup \text{all-acquired (sb @ [Read_{sb} volatile a t v])} \subseteq O_{sb} \cup \text{all-acquired sb}$
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: $ts_{sb}' O_{sb}'$)
next
case False
show ?thesis
proof (unfold-locales)
  fix n m
  fix $p_{n} i_{n} O_{n} R_{n} \varnothing_{n} D_{n} \varnothing_{m} O_{m} R_{m} D_{m} \varnothing_{m} S_{m}$
  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: $ts_{sb}'!n = (p_{n}, i_{n}, O_{n}, R_{n}, D_{n}, O_{n}, R_{n})$
  and mth: $ts_{sb}'!m = (p_{m}, i_{m}, O_{m}, R_{m}, D_{m}, O_{m}, R_{m})$
  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' $O_{sb}'$)

show $(O_{m} \cup \text{all-acquired sb}_{m}) \cap \text{read-only-reads (acquired True (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_{n}) O_{n})}$
  (dropWhile (Not ◦ 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': $ts_{sb}'!n = (p_{n}, i_{n}, \varnothing_{n}, D_{n}, O_{n}, R_{n})$
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 acq-eq

ultimately show ?thesis

using True ts_{sb}-i nth mth n-bound'm-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}, 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

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 mth n-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 ts_{sb}' sb'O_{sb}')+

done

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

next

show ownership-distinct ts_{sb}'

proof –

have all-acquired (sb @ [Read_{sb} volatile a t v]) \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 hcons: history-consistent \( \theta_{sb} \) (hd-prog \( p_{sb} \)) \( sb \).
from load-tmps-read-tmps-distinct [OF i-bound ts\sb-i]
have t-notin-reads: \( t \notin \text{read-tmps} \) \( sb \)
  by (auto simp add: is\sb)
from load-tmps-write-tmps-distinct [OF i-bound ts\sb-i]
have t-notin-writes: \( t \notin \bigcup \text{write-sops} \) \( sb \)
  by (auto simp add: is\sb)
from valid-write-sops [OF i-bound ts\sb-i]
have valid-sops: \( \forall \text{sop} \in \text{write-sops} \) \( sb \)
  by auto
from load-tmps-fresh [OF i-bound ts\sb-i]
have t-fresh: \( t \notin \text{dom} \) \( \theta_{sb} \)
  using is\sb
  by simp
have history-consistent \((\theta_{sb}(t \mapsto v))\)
  (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-append\II)
  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′ \( \theta_{sb}′ \))
  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

447
have sharing-consistent $s_{sb}' \cap O_{sb}'$ (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 $s_{sb}' \cap ts_{sb}'$
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 $s_{sb}' \cap ts_{sb}'$
by (simp add: $s_{sb}' \cap ts_{sb}' \cap O_{sb}'$
next
from unowned-shared-nth-update [OF i-bound ts$_{sb}-i$ subset-refl]
show unowned-shared $s_{sb}' \cap ts_{sb}'$
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 (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 $s_{sb}' \cap ts_{sb}'$
by (simp add: $s_{sb}' \cap ts_{sb}'$
qed

have tmps-distinct $t$: 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 v])
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$_{sb}-i$]
load-tmps-distinct [OF i-bound ts$_{sb}-i$]
have load-tmps is$_{sb}' \cap$ read-tmps (sb @ [Read$_{sb}$ volatile a t v]) = {}
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$_{sb}'$
qed

448
have valid-sops': valid-sops ts_{sb}'
proof -
from valid-store-sops [OF i-bound ts_{sb}-i]
have valid-store-sops': \forall sop \in 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 write-sops (sb@ [Read_{sb} volatile a t v]), 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 \theta_{sb}') is_{sb}'
  by (auto simp add: is_{sb} \theta_{sb}')
from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]
have load-tmps is_{sb}' \cap \bigcup (fst ' write-sops (sb@ [Read_{sb} volatile a t v])) = \{\}
  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}') \cap dom \theta_{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 (\theta_{sb}(t \mapsto v)) = \{\}
  by auto
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis by (simp add: ts_{sb}' 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@ [Read_{sb} volatile a t v]) = \{\}
  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 
   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 
   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\_′)
qed

show ?thesis 
proof 
  (cases outstanding-refs is-volatile-Write 
   \{\} 
   )
  case True 
  from True have flush-all: takeWhile (Not \circ is-volatile-Write\_sb) sb = sb 
   by (auto simp add: outstanding-refs-conv )
  from True have suspend-nothing: dropWhile (Not \circ is-volatile-Write\_sb) sb = [] 
   by (auto simp add: outstanding-refs-conv)
  hence suspends-empty: suspends = [] 
   by (simp add: suspends-empty)
  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: tsl! = (p\_sb, Read volatile a t \ # is\_sb\′, \_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\′, \_sb, (), m, D, acquired True ?take-sb O\_sb, 
   release ?take-sb (dom S\_sb \, R\_sb, S)) \rightarrow 
   (is\_sb\′, \_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 (ts[\_i := (p\_sb, is\_sb\′, \_sb (t \mapsto m a), (), D, acquired True ?take-sb O\_sb, release ?take-sb (dom S\_sb \, 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\′, 
   \_sb (t \mapsto v), sb @ [Read\_sb volatile a t v], D\_sb, O\_sb, R\_sb)]) m\_sb = 
   flush-all-until-volatile-write ts\_sb m\_sb.
from True witness have not-volatile\′: volatile\′ = False

450
by (auto simp add: outstanding-refs-conv)

from witness not-volatile' have a-out-sb: a ∈ outstanding-refs (Not o is-volatile) sb
apply (cases sop')
apply (fastforce simp add: outstanding-refs-conv is-volatile-def split: memref.splits)
done

with non-volatile-owned-or-read-only-outstanding-refs
[OF outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb}-i]]
have a-owned: a ∈ O_{sb} ∪ all-acquired sb ∪ read-only-reads O_{sb} sb
by auto

have flush-all-until-volatile-write ts_{sb} m_{sb} a = v
proof –
  have ∀ j < length ts_{sb}. i ≠ j ⟹
    (let (·,·,·,sb_j,·,·,·) = ts_{sb}[j]
      in a ∉ outstanding-refs is-non-volatile-Write_{sb}
      (takeWhile (Not o is-volatile-Write_{sb}) sb_j))
  proof –
  { fix j p_j is_j R_j D_j xs_j sb_j
  assume j-bound: j < length ts_{sb}
  assume neq-i-j: i ≠ j
  assume jth: ts_{sb}[j] = (p_j,is_j,xs_j, sb_j, D_j, O_j, R_j)
  have a ∉ outstanding-refs is-non-volatile-Write_{sb}
    (takeWhile (Not o is-volatile-Write_{sb}) sb_j)
  proof
  let ?take-sb_j = (takeWhile (Not o is-volatile-Write_{sb}) sb_j)
  let ?drop-sb_j = (dropWhile (Not o is-volatile-Write_{sb}) sb_j)
  assume a-in: a ∈ outstanding-refs is-non-volatile-Write_{sb} ?take-sb_j
  with outstanding-refs-takeWhile [where P'≡ Not o is-volatile-Write_{sb}]
  have a-in': a ∈ outstanding-refs is-non-volatile-Write_{sb} sb_j
  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 jth]]
  have j-owns: a ∈ O_j ∪ all-acquired sb_j
  by auto
  with ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i jth]
  have a-not-owns: a ∉ O_{sb} ∪ all-acquired sb
  by blast

from non-volatile-owned-or-read-only-append [of False \mathcal{S}_{sb} O_j ?take-sb_j ?drop-sb_j]
  outstanding-non-volatile-refs-owned-or-read-only [OF j-bound jth]
have non-volatile-owned-or-read-only False \mathcal{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 ∈ O_j ∪ all-acquired ?take-sb_j
by auto

451
  have no-unsharing:release ?take-sb \_j (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 acquired-all-acquired-in [OF this] ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i-jth]
  j-owns
  have False
  by auto
\}
moreover
\{
  assume a-ro: a \in \text{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 \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 jth 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 jth]
  non-volatile-writes-unshared-append [of S_{sb} ?take-sb j ?drop-sb]
  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 ∧ a \in \text{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} ?take-sb j ?drop-sb]
    have unshared-take: non-volatile-writes-unshared S_{sb} (takeWhile (Not \circ is-volatile-Write_{sb}) 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 ?\( \text{take-sb} \) \( \mathcal{O}_{\text{sb}} \)) ?\( \text{drop-sb} \)
  by (simp add: \( S \))

from sharing-consistent-share-all-shared

have dom (share ?\( \text{drop-sb} \) \( S \)) \( \subseteq \) dom \( S \cup \) all-shared ?\( \text{drop-sb} \)
  by auto

also from sharing-consistent-all-shared [OF sc]

have \( \ldots \) \( \subseteq \) dom \( S \cup \) acquired True ?\( \text{take-sb} \) \( \mathcal{O}_{\text{sb}} \) by auto

also from acquired-all-acquired all-acquired-takeWhile

have \( \ldots \) \( \subseteq \) dom \( S \cup (\mathcal{O}_{\text{sb}} \cup \) all-acquired \( \text{sb} \) \) by force

finally

have a-shared: \( a \in \) 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: \( a \in \) dom (share ?\( \text{take-sb} \_j \) \( S_{\text{sb}} \))
    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

hence m-a-v: \( m \ a = v \)
  by (simp add: m)

have tmps-commute: \( \hat{\theta}_{\text{sb}}(t \mapsto v) = (\hat{\theta}_{\text{sb}} \upharpoonright (\text{dom } \hat{\theta}_{\text{sb}} - \{t\}))(t \mapsto v) \)
  apply (rule ext)
  apply (auto simp add: restrict-map-def domIff)
  done

from suspend-nothing

have suspend-nothing': (dropWhile (Not \( \circ \) is-volatile-Write\( s_{\text{sb}} \)) \( s_{\text{b}}' \)) = []
by (simp add: sb')

from D
have D': D_sb = (D ∨ outstanding-refs is-volatile-Write sb (sb@[Read sb volatile a t v]) ≠ {}) by (auto simp: outstanding-refs-append)

have (ts_sb', m_sb, S_sb') ∼ (ts[i := (p sb, is sb')], D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb), m, S)
  apply (rule sim-config.intros)
  apply (simp add: m flush-commute ts sb' O sb' θ 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' θ sb' R sb')
  using leq
  apply (simp add: ts sb' sb' O sb' S sb' θ sb' R sb')
  using i-bound i-bound' ts-sim ts-i True D sb'
  apply (clarsimp simp add: Let-def nth-list-update outstanding-refs-conv m-a-v ts sb' O sb' θ 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 ?thesis
  using valid-own' valid-list' 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: 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 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'#
  and
  drop: dropWhile (Not o is-volatile-Write sb) sb = Write sb True a' sop' v' A' L' R' W'#
  subgoal for y ys
  apply (case-tac y)
  apply auto
done

done

454
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

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’, θ sb(t :=→ v), D sb, O sb, R sb)]) =
m sb =
flush-all-until-volatile-write ts sb m sb.

have Write sb True a’ sop’ v’ A’ L’ R’ W’ ∈ set sb
by (subst sb-split) auto

from dropWhile-append1 [OF this, of (Not ◦ is-volatile-Write sb)]
have drop-app-comm:
(dropWhile (Not ◦ is-volatile-Write sb) (sb @ [Read sb volatile a t v])) =
dropWhile (Not ◦ is-volatile-Write sb) sb @ [Read sb volatile a t v]
by simp

from load-tmps-fresh [OF i-bound ts sb-i]
have t /∈ dom θ sb
by (auto simp add: is sb)
then have tmss-commute:
θ sb |′ (dom θ sb – read-tmps sb”’) =
θ sb |′ (dom θ 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 v]) ≠ {})
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’ 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’ θ sb’)
using leq
apply (simp add: ts sb’)
using i-bound i-bound’ ts-sim ts-i is-sim D’

455
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
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’ tmpps-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 = Read volatile a t ≠ is sb’ and
O sb’: O sb’=O sb and
R sb’: R sb’=R sb and
θ sb’: θ sb’ = θ sb (t→(m sb a)) and
sb’: sb’=sb@[Read sb 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 ∈ 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

{ fix j p j is sb j O j R j D sb j θ sb j sb j
assume j-bound: j < length ts sb
assume neq-i-j: i ≠ j
assume jth: ts sb!j = (p j, is sb j, θ sb j, sb j, D sb j, O j, R j)
assume non-vol: ¬ volatile
have a /∈ O j ∪ all-acquired sb j

456
proof
assume a-j: a ∈ O_j ∪ all-acquired sb_j
let ?take-sb_j = (takeWhile (Not o is-volatile-Write sb_j) sb_j)
let ?drop-sb_j = (dropWhile (Not o is-volatile-Write sb_k) 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_j = is_j @ prog-instrs suspends_j and
D_j: D_{sb_j} = (D_j ∪ outstanding-refs is-volatile-Write sb_j ∉ {}) and
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}) R_j)
by (auto simp add: Let-def)

from a-j ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb-i j}]
have a-notin-sb: a ∉ O_{sb} ∪ all-acquired sb
by auto
with acquired-all-acquired [of True sb O_{sb}]
have a-not-acq: a ∉ acquired True sb O_{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 O_{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-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 ∉ dom R_j and
a-shared_j: a ∉ all-shared ?take-sb_j
by auto

have a ∉ ∪ ((λ(−, −, −, sb, −, −, −). all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb)) ')
set ts_{sb}:
proof

fix k p_k is_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, ∞ sb_k, D_k, O_k, R_k)
assume a-in: a ∈ all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb_k)

457
have False
proof (cases k=j)
case True with a-shared 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 \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_k}) sb_k
dropWhile (Not \circ \text{is-volatile-Write}_{sb_k}) sb_k]
all-shared-append [of takeWhile (Not \circ \text{is-volatile-Write}_{sb_k}) sb_k
dropWhile (Not \circ \text{is-volatile-Write}_{sb_k}) 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 \in \text{read-only S sb}
by (auto simp add: S)

have own-dist: ownership-distinct ts sb by fact
have share-consis: sharing-consis 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 S (acquired True ?take-sb O sb) ?drop-sb
by (simp add: 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 S sb)}
by (auto simp add: domIff S 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 ?take-sb j ?drop-sb j]
have weak-consis-drop:weak-sharing-consistent O j ?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 \text {all-acquired ?take-sb j}
by auto
with a-j a-notin-owns-j
have a-drop: a \in \text{all-acquired ?drop-sb j}
using all-acquired-append [of ?take-sb j ?drop-sb j]

458
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 ∃ sop a' v ys zs A L R W.
  (suspends_j = ys @ Write sb 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 A L R W# zs
    (is suspends_j = ?suspends) and
    a-A': a ∈ A'
  by blast

from sharing-consis [OF j-bound jth]
have sharing-consistent S sb O_j sb_j.
then have A'-R': A' ∩ R' = {}
  by (simp add: sharing-consistent-append [of - - ?take-sb 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.
then have cph: causal-program-history is sbj ?suspends
  apply −
  apply (rule causal-program-history-suffix [where sb=?take-sb 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) 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 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 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-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 $\cap$ 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 $?take-sb_j O_j$) (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@[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$_j$@??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 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)
  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 $p_j$ $sb_j$) ??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

460
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 \(\theta_{sbj}\) (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@[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
\(S=\text{share} ?\text{drop-sb} \ S\)

obtain is\(_j\)' \(R_j\)' where
is\(_j\)' \(\in\) instrs zs @ is\(_sbj\) = is\(_j\)' \(\in\) 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']),
is\(_j\)'',
\(\theta_{sbj}\)' (dom \(\theta_{sbj}\) \(-\) read-tmps zs),
(), True, acquired True ys (acquired True ?take-sb\(_j\) \(O_j\)) \(\cup\) A' \(-\) R',\(R_j\)'\),
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 (-,-,r) \(\Rightarrow d^*\) (?ts-ys,?m-ys,?shared-ys))

461
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-ys!i = (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 ∈ 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

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} A L R W # zs ∧ a ∈ A
then
obtain A′ L′ R′ W′ ys zs where
    split-suspends_j: suspends_j = ys @ Ghost_{sb} 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_{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′_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_{sbj}) 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

462
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 jth]
have reads-consis-j: reads-consistent False \(\mathcal{O}_j\) \(m_{sb}\) \(sb\).
from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing \(S_{sb}\) \(ts_{sb}\) j-bound]
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} i jth]
have outstanding-refs is-Write_{sb} \(?drop-sb\) \(\cap\) outstanding-refs is-non-volatile-Read_{sb} suspends_j = {}
by (simp add: \(m\) suspends_j)

hence reads-consis-ys: reads-consistent True (acquired True \(?take-sb\) \(O_j\)) (flush \(?drop-sb\) \(m\) suspends_j).

from valid-write-sops [OF j-bound jth]
have \(\forall\) sop\(\in\)write-sops (\(?take-sb\) \(\cap\) suspends_j \(\cap\) reads-consistent-append)
by (simp add: split-suspends_j [symmetric] suspends_j)
then obtain valid-sops-take: \(\forall\) sop\(\in\)write-sops \(?take-sb\). valid-sop sop and valid-sops-drop: \(\forall\) sop\(\in\)write-sops (ys@\(\text{Ghost}_{sb} A' L' R' W'\)). valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound jth]
have distinct-read-tmps (\(?take-sb\) \(\cap\) suspends_j)
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 jth]
have h-consis:
history-consistent \(\emptyset_{sb}\) (hd-prog \(p_j\) (\(?take-sb\) \(\cap\) suspends_j)) (\(?take-sb\) \(\cap\) 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-cons] 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-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=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' j, flush ?drop-sb m, share ?drop-sb S) \Rightarrow^* 
(\?ts j' j':=(last-prog
(hd-prog p j (Ghost sb A L' R' W'\# zs)) (ys@[Ghost sb A L' R' W'])
\is j',
464
\( \vartheta_{sbj} \vdash (\text{dom} \vartheta_{sbj} - \text{read-tmps} \text{zs}), \)

\( \vartheta_{sbj} \), \( \mathcal{D}_{sbj} \lor \text{outstanding-refs \ is-volatile-Write}_{sbj} \ (\text{ys} @ [\text{Ghost}_{sbj} A' L' R' W']) \neq \{\}, \)

\( \text{acquired} \ \text{True} \ \text{ys} \ (\text{acquired} \ \text{True} \ \text{?take-sb} \ \text{O}_{sbj} \cup A' - R', R')], \)

\( \text{flush} \ (\text{ys}@([\text{Ghost}_{sbj} A' L' R' W']) \ (\text{flush} \ ?\text{drop-sb} \ m), \)

\( \text{share} \ (\text{ys}@([\text{Ghost}_{sbj} A' L' R' W']) \ (\text{share} \ ?\text{drop-sb} \ S)) \)

\( \text{(is} \ (-,+) \Rightarrow d^* \ (?\text{ts-ys}, ?\text{m-ys}, ?\text{shared-ys}) \) \)

\( \text{by} \ (\text{auto simp add: acquired-append}) \)

\textbf{from} \ \text{i-bound'} \ \textbf{have} \ \text{i-bound-ys: \ i < length} \ ?\text{ts-ys} \)

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

\textbf{from} \ \text{i-bound'} \ \text{neq-i-j} \)

\textbf{have} \ ?\text{ts-ys-i} \ : \ ?\text{ts-ys}!i = (p_{sbj}, i_{sbj}, \vartheta_{sbj}, (), \ D_{sbj}, \ \text{acquired} \ \text{True} \ \text{sb} \ \text{O}_{sbj}, \ \text{release} \ \text{sb} \ (\text{dom} \ \mathcal{S}_{sbj}) \ \mathcal{R}_{sbj}) \)

\( \text{by} \ \text{simp} \)

\textbf{note} \ \text{conflict-computation} = \ rtranclp-trans \ [\text{OF} \ \text{steps-flush-sb} \ \text{steps-ys}] 

\textbf{from} \ \text{safe-reach-safe-rtrancl} \ [\text{OF} \ \text{safe-reach} \ \text{conflict-computation}] \)

\textbf{have} \ \text{safe-delayed} \ (?\text{ts-ys}, ?m-ys, ?shared-ys). 

\textbf{from} \ \text{safe-delayedE} \ [\text{OF} \ \text{this} \ \text{i-bound-ys} \ ?\text{ts-ys-i}, \ \text{simplified} \ \text{is}_{sbj} \] \ \text{non-vol} \ \text{a-not-acq} 

\textbf{have} \ a \in \ \text{read-only} \ (\text{share} \ (\text{ys}@([\text{Ghost}_{sbj} A' L' R' W']) \ (\text{share} \ ?\text{drop-sb} \ S)) \)

\( \text{apply} \ \text{cases} \)

\( \text{apply} \ (\text{auto simp add: Let-def \ is}_{sbj}) \)

\textbf{done} 

\textbf{with} \ a-A'

\textbf{show} \ False 

\( \text{by} \ (\text{simp add: share-append in-read-only-convs}) \)

\textbf{qed}

\textbf{qed} 

\textbf{note} \ \text{non-volatile-unowned-others} = \ \text{this} 

{} 

\textbf{assume} \ a-in: \ a \in \ \text{read-only} \ (\text{share} \ (\text{dropWhile} \ (\text{Not} \ \circ \ \text{is-volatile-Write}_{sbj}) \ \text{sb}) \ \mathcal{S}) \)

\textbf{assume} \ nv: \ \neg \ \text{volatile} \)

\textbf{have} \ a \in \ \text{read-only} \ (\text{share} \ \text{sb} \ \mathcal{S}_{sbj}) \)

\textbf{proof} \ (\text{cases} \ a \in \ \mathcal{O}_{sbj} \cup \ \text{all-acquired} \ \text{sb}) 

\textbf{case} \ \text{True} 

\textbf{from} \ \text{share-all-until-volatile-write-thread-local'} \ [\text{OF} \ \text{ownership-distinct-ts}_{sbj} 

\text{sharing-consis-ts}_{sbj} \ \text{i-bound} \ \text{ts}_{sbj}-i \ \text{True} \] \ \text{True} \ \text{a-in} 

\textbf{show} \ \text{?thesis} 

\( \text{by} \ (\text{simp add: \ i-bound \ \text{ts}_{sbj}-i \ True}) \)

\textbf{next} 

\textbf{case} \ \text{False} 

\textbf{from} \ \text{read-only-share-unowned} \ [\text{OF} \ \text{weak-consis-drop} - a-in] \ \text{False} 

\textbf{acquired-all-acquired} \ [\text{of} \ \text{True} \ ?\text{take-sb} \ \mathcal{O}_{sbj}] \ \text{all-acquired-append} \ [\text{of} \ ?\text{take-sb} \ ?\text{drop-sb}] 

465
have a-ro-shared: \( a \in \text{read-only } S \)
by auto
have \( a \notin \bigcup ((\lambda (-, -, -, sb, -, -, -)) \cdot \text{all-shared (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) sb)) \cdot \text{set } ts_{sb} )
proof 
{
  fix k \ p_k \ i_s k \ u_k \ s k \ D_k \ O_k \ R_k 
  assume k-bound: k < \text{length } ts_{sb} 
  assume ts-k: ts_{sb} ! k = (p_k,i_s k,u_k,s k,D_k,O_k,R_k) 
  assume a-in: a \in \text{all-shared (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) sb_k)}
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 \circ \text{is-volatile-Write}_{sb} ) sb_k]
  dropWhile (Not \circ \text{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 
  have a \notin \text{dom (release (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) sb_k)} \cdot (\text{dom (} S_{sb}\text{)})
  by (auto simp add: Let-def) 
  with dom-release-takeWhile [of sb_k \cdot (\text{dom (} S_{sb}\text{)}) \cdot R_k]
  obtain
    a-rels: a \notin \text{dom } R_k \text{ and}
    a-shared_r: a \notin \text{all-shared (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) sb_k)}
  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 \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 
}
note non-vol-ro-reduction = this

have valid-own': valid-ownership \( S_{sb}' \cdot ts_{sb}' \)
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only \( S_{sb}' \cdot ts_{sb}' \)
proof (cases volatile)

466
\textbf{case} False \nl \textbf{from} outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts\textsubscript{sb}-i] \nl \textbf{have} non-volatile-owned-or-read-only False \( S_{sb} \ O_{sb} \).
\nl \textbf{then} \nl
\textbf{have} non-volatile-owned-or-read-only False \( S_{sb} \ O_{sb} \ (sb@[\text{Read}_{sb} \ False \ a \ t \ (m_{sb} \ a)]) \) \nl \textbf{using} access-cond' False non-vol-ro-reduction \nl \textbf{by} (auto simp add: non-volatile-owned-or-read-only-append) 

\textbf{from} outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this] \nl \textbf{show} ?thesis \textbf{by} (auto simp add: False ts\textsubscript{sb}' O_{sb}' S_{sb}')

\textbf{next} \nl \textbf{case} True \nl \textbf{from} outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts\textsubscript{sb}-i] \nl \textbf{have} non-volatile-owned-or-read-only False \( S_{sb} \ O_{sb} \).
\nl \textbf{then} \nl
\textbf{have} non-volatile-owned-or-read-only False \( S_{sb} \ O_{sb} \ (sb@[\text{Read}_{sb} \ True \ a \ t \ (m_{sb} \ a)]) \) \nl \textbf{using} True \nl \textbf{by} (simp add: non-volatile-owned-or-read-only-append) 

\textbf{from} outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this] \nl \textbf{show} ?thesis \textbf{by} (auto simp add: True ts\textsubscript{sb}' O_{sb}' S_{sb}')

\textbf{qed}

\textbf{next} \nl \textbf{show} outstanding-volatile-writes-unowned-by-others ts\textsubscript{sb}'

\textbf{proof} \textbf{−} \nl \textbf{have out: outstanding-refs is-volatile-Write}_{sb} (sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)]) \subseteq outstanding-refs is-volatile-Write_{sb} \nl \textbf{by} (auto simp add: outstanding-refs-append) 
\textbf{have all-acquired (sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)]) \subseteq all-acquired sb} \nl \textbf{by} (auto simp add: all-acquired-append) 
\textbf{from outstanding-volatile-writes-unowned-by-others-store-buffer} 
[OF i-bound ts\textsubscript{sb}-i out this] \nl \textbf{show} ?thesis \textbf{by} (simp add: ts\textsubscript{sb}' O_{sb}')

\textbf{qed}

\textbf{next} \nl \textbf{show} read-only-reads-unowned ts\textsubscript{sb}'

\textbf{proof} (cases volatile) \nl \textbf{case} True \nl \textbf{have t: read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write_{sb})} 
(sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)])) O_{sb}) \nl \textbf{by} (auto simp add: outstanding-refs-append) 
\textbf{have all-acquired (sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)]) \subseteq all-acquired sb} \nl \textbf{by} (auto simp add: all-acquired-append) 
\textbf{from outstanding-volatile-writes-unowned-by-others-store-buffer} 
[OF i-bound ts\textsubscript{sb}-i out this] \nl \textbf{show} ?thesis \textbf{by} (simp add: ts\textsubscript{sb}' O_{sb}')

\textbf{qed}

\textbf{next} \nl \textbf{show} read-only-reads-unowned ts\textsubscript{sb}'

\textbf{proof} (cases volatile) \nl \textbf{case} True \nl \textbf{have t: read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write_{sb})} 
(sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)])) O_{sb}) \nl \textbf{by} (auto simp add: outstanding-refs-append) 
\textbf{have all-acquired (sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)]) \subseteq all-acquired sb} \nl \textbf{by} (auto simp add: all-acquired-append) 
\textbf{from outstanding-volatile-writes-unowned-by-others-store-buffer} 
[OF i-bound ts\textsubscript{sb}-i out this] \nl \textbf{show} ?thesis \textbf{by} (simp add: ts\textsubscript{sb}' O_{sb}')

\textbf{qed}

\textbf{show} read-only-reads-unowned ts\textsubscript{sb}'

\textbf{proof} (cases volatile) \nl \textbf{case} True \nl \textbf{have t: read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write_{sb})} 
(sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)])) O_{sb}) \nl \textbf{by} (auto simp add: outstanding-refs-append) 
\textbf{have all-acquired (sb @ [\text{Read}_{sb} volatile a t (m_{sb} a)]) \subseteq all-acquired sb} \nl \textbf{by} (auto simp add: all-acquired-append) 
\textbf{from outstanding-volatile-writes-unowned-by-others-store-buffer} 
[OF i-bound ts\textsubscript{sb}-i out this] \nl \textbf{show} ?thesis \textbf{by} (simp add: ts\textsubscript{sb}' O_{sb}')

\textbf{qed}

\textbf{next}
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\_i} \bigcup all-acquired sb)

next
  case False
  show ?thesis
  proof (unfold-locales)
    fix n m
    fix p_n \is_n \O_n \R_n \D_n \emptyset_n \sb_n \text{ i-bound: } n < \text{ length ts}_{sb\_i}'
    and m-bound: m < \text{ length ts}_{sb\_i}'
    and neq-n-m: n \neq m
    and nth: ts_{sb\_i}'!n = (\text{p}_n, \text{is}_n, \emptyset_n, \sb_n, \D_n, \O_n, \R_n)
    and mth: ts_{sb\_i}'!m = (\text{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\_i}'
    from m-bound have m-bound\': m < \text{ length ts}_{sb\_i}'
    have acq-eq: (\text{O}_{sb\_i}' \cup \text{all-acquired sb}') = (\text{O}_{sb} \cup \text{all-acquired sb})
      by (simp add: all-acquired-append sb_{sb\_i}')
    show (\text{O}_{m} \cup \text{all-acquired sb\_m}) \cap
      \text{read-only-reads (acquired True (takeWhile (Not \is-volatile-Write_{sb\_i}) \sb_n) \O_n)}
      (\text{dropWhile (Not \is-volatile-Write_{sb\_i}) \sb_n}) = {}
    proof (cases m=i)
      case True
      with neq-n-m have neq-n-i: n \neq i
      by (auto simp add: ts_{sb\_i} \bigcup all-acquired sb)
      note read-only-reads-unowned [OF i-bound m-bound\' \bigcup all-acquired sb\_i]
    moreover
      note acq-eq
    ultimately show ?thesis
    using True ts_{sb\_i}-i nth mth n-bound\' m-bound'
    by (simp add: ts_{sb\_i}')
  next
    case False
    note neq-m-i = this
    with m-bound mth i-bound have mth\': ts_{sb\_i}!m = (\text{p}_m, \text{is}_m, \emptyset_m, \sb_m, \D_m, \O_m, \R_m)
    by (auto simp add: ts_{sb\_i}')
    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}-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}’ nth nth m-bound’ m-bound’ neq-m-i
apply (case-tac outstanding-ref (is-volatile-Write_{sb}) sb = { })
apply (clarsimp simp add: outstanding-vol-write-take-drop-append
  acquired-append read-only-reads-append ts_{sb}’ sb’ ()
  done
next
case False
with n-bound nth i-bound have nth’: ts_{sb}’ n = (p_n, is_n, θ_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
show ownership-distinct ts_{sb}’
proof –
  have all-acquired (sb @ [Read_{sb} volatile a t (m_{sb} a)]) ⊆ 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]
  show ?thesis by (simp add: ts_{sb}’ sb’ ()
qed
qed

have valid-hist’ valid-history program-step ts_{sb}’
proof –
from valid-history [OF i-bound ts_{sb}’]
have hcons: history-consistent θ_{sb} (hd-prog p_{sb} sb) sb.
from load-tmps-read-tmps-distinct [OF i-bound ts_{sb}’]
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}’]
have t-notin-writes: t ∉ ∪ (fst ‘ write-sops sb )
  by (auto simp add: is_{sb})
from valid-write-sops [OF i-bound ts_{sb}’]
have valid-sops: ∀ sop ∈ write-sops sb. valid-sop sop
  by auto
from load-tmps-fresh [OF i-bound ts_{sb}’]
have t-fresh: t ∉ dom θ_{sb}
  using is_{sb}
  by simp

469
from valid-implies-valid-prog-hd [OF i-bound ts\sb-i valid]

have history-consistent (\(\emptyset\sb_{\text{t}} \mapsto \text{m}_{\text{sb}}\ a\))
  (hd-prog \(\text{p}_{\text{sb}}\) (\(\text{sb}@\ [\text{Read}_{\text{sb}}\ \text{volatile}\ a\ \text{t} \ (\text{m}_{\text{sb}}\ a)]\))
  (\(\text{sb}@\ [\text{Read}_{\text{sb}}\ \text{volatile} \ a \ (\text{m}_{\text{sb}}\ a)]\))
using t-notin-writes valid-sops t-fresh hcons
apply –
apply (rule history-consistent-appendI)
apply (auto simp add: hd-prog-append-Read)
done

from valid-history-nth-update [OF i-bound this]
show ?thesis
  by (auto simp add: ts\sb′ sb′ \(\emptyset\sb_{\text{t}}\sb′\))
qed

from reads-consistent-unbuffered-snoc [OF buf-None refl valid-reads [OF i-bound ts\sb-i]
volatile-cond]
  have reads-consis'; reads-consistent False \(\text{O}_{\text{sb}}\ \text{m}_{\text{sb}}\ (\text{sb}@\ [\text{Read}_{\text{sb}}\ \text{volatile} \ a \ (\text{m}_{\text{sb}}\ a)])\)
  by (simp split: if-split-asm)

from valid-reads-nth-update [OF i-bound this]
have valid-reads': valid-reads \(\text{m}_{\text{sb}}\ \text{ts}_{\text{sb}}\ \text{ts}_{\text{sb}}\)
  by (simp add: ts\sb′ sb′ \(\emptyset\sb_{\text{t}}\sb′\))

have valid-sharing': valid-sharing \(\text{S}_{\text{sb}}\ \text{ts}_{\text{sb}}\)
proof (intro-locales)
from outstanding-non-volatile-writes-unshared [OF i-bound ts\sb-i]
have non-volatile-writes-unshared \(\text{S}_{\text{sb}}\) (\(\text{sb}@\ [\text{Read}_{\text{sb}}\ \text{volatile} \ a \ (\text{m}_{\text{sb}}\ a)]\))
  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 \(\text{S}_{\text{sb}}\ \text{ts}_{\text{sb}}\)
  by (simp add: ts\sb′ sb′ \(\text{S}_{\text{sb}}\))
next
from sharing-consis [OF i-bound ts\sb-i]
have sharing-consistent \(\text{S}_{\text{sb}}\ \text{O}_{\text{sb}}\ \text{sb}\.\)
then
have sharing-consistent \(\text{S}_{\text{sb}}\ \text{O}_{\text{sb}}\) (\(\text{sb}@\ [\text{Read}_{\text{sb}}\ \text{volatile} \ a \ (\text{m}_{\text{sb}}\ a)]\))
  by (simp add: sharing-consistent-append)
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis \(\text{S}_{\text{sb}}\ \text{ts}_{\text{sb}}\)
  by (simp add: ts\sb′ \(\text{O}_{\text{sb}}\ \text{sb}' \text{S}_{\text{sb}}\)')
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 \(\text{S}_{\text{sb}}\ \text{ts}_{\text{sb}}\)
  by (simp add: \(\text{S}_{\text{sb}}\ \text{ts}_{\text{sb}}\ \text{O}_{\text{sb}}\)')
next
from unowned-shared-nth-update [OF i-bound ts\sb-i subset-refl]
show unowned-shared \(\text{S}_{\text{sb}}\ \text{ts}_{\text{sb}}\) by (simp add: \(\text{S}_{\text{sb}}\ \text{O}_{\text{sb}}\ \text{S}_{\text{sb}}\'))
next

470
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 (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 /∈ 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 load-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}' ∩ 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 \( \theta_{sb}' \)) \( is_{sb}' \)
by (auto simp add: \( is_{sb} \) \( \theta_{sb} \))

from load-tmps-write-tmps-distinct [OF i-bound ts sb -i]
have load-tmps \( is_{sb}' \cap \bigcup (\{ \text{fst \ ' write-sops (sb@ [Read \_ \_ 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 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 (\( \text{Read volatile a t} \# is_{sb}' \)) \( \cap \) dom \( \theta_{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 \text{dom (} (\theta_{sb}(t \mapsto (m_{sb} a)) \text{)} = \{ \}
by auto

from load-tmps-fresh-nth-update [OF i-bound this]
show \( ? \)thesis by (simp add: ts sb sb sb sb sb sb)
qed

have enough-flushs': enough-flushs ts sb'
proof

from clean-no-outstanding-volatile-Write sb [OF i-bound ts sb -i]
have \( (D_{sb} \longrightarrow \text{outstanding-refs is-volatile-Write}_{sb} (sb@[Read}_{sb} \text{volatile a t (m}_{sb} a)] \) = \{ \}
by (auto simp add: outstanding-refs-append )

from enough-flushs-nth-update [OF i-bound this]
show \( ? \)thesis by (simp add: ts sb sb sb sb sb 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 \( is_{sb}' (sb@[Read}_{sb} \text{volatile a t (m}_{sb} a)] \)
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 \( p_{sb} (sb@[Read}_{sb} \text{volatile a t (m}_{sb} a)] = p_{sb} \)
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 sb)
qed
show ?thesis
proof (cases outstanding-refs is-volatile-Write sb = { })
case True

from True have flush-all: takeWhile (Not o 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 = Read volatile a t # is sb' by (simp add: is sb)

with suspends-empty ts-i have ts-i: ts[i := (p sb, is sb', vs 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', vs sb, (), m, \ D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb)] by simp
from direct-computation.concurrent-step.Memop [OF i-bound ts sb -i [simplified is sb]] have flush-commute: flush-all-until-volatile-write ts sb m sb a = m sb a by simp

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', vs sb, (m sb a)], sb @ [Read sb volatile a t (m sb a)], \ D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb)] by simp

have flush-all-until-volatile-write ts sb m sb a = m sb a

proof 
  have \forall j < length ts sb, i \neq j \longrightarrow
  (let (\cdot,\cdot,\cdot, sb j,\cdot,\cdot,\cdot) = ts sb[i]
  in a \notin outstanding-refs is-non-volatile-Write sb (takeWhile (Not o is-volatile-Write sb) sb j))

proof 
  { 
    fix j p j is j \ O j \ D j acq j x j sb j
    assume j-bound: j < length ts sb
    assume neq-i-j: i \neq j
    assume jth: ts sb[i] = (p j, is j, x j, sb j, \ D j, O j, \ R j)

    473
have a \notin \text{outstanding-refs} \text{is-non-volatile-Write}_{sbj} (\text{takeWhile} (\text{Not} \text{is-volatile-Write}_{sbj}) \text{sbj})

proof

let ?take-sb_j = (\text{takeWhile} (\text{Not} \text{is-volatile-Write}_{sbj}) \text{sbj})
let ?drop-sb_j = (\text{dropWhile} (\text{Not} \text{is-volatile-Write}_{sbj}) \text{sbj})
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} \text{is-volatile-Write}_{sb_j}
have a-in': a \in \text{outstanding-refs} \text{is-non-volatile-Write}_{sb_j} \text{sbj}
  by auto

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}]
have j-owner: a \in O_j \cup \text{all-acquired} \text{sbj}
  by auto

with \text{ownership-distinct} \text{OF i-bound j-bound neq-i-j ts}_{sb-i jth}
have a-not-owns: a \notin O_{sb} \cup \text{all-acquired}
  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-bound jth}
have 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-in
have j-owns-drop: a \in O_j \cup \text{all-acquired} ?take-sb_j
  by auto

from \text{rels-cond} \text{rule-format, OF j-bound [simplified leq neq-i-j] ts-sim}
  \text{OF j-bound} jth
have 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 \text{sb} O_{sb}
  with \text{acquired-all-acquired-in} \text{OF this} \text{ownership-distinct} \text{OF i-bound j-bound neq-i-j ts}_{sb-i jth}
  j-owns
  have False
  by auto
\}
moreover
\{  
  assume a-ro: a \in \text{read-only} \text{share} ?drop-sb S
  from \text{read-only-share-unowned-in} \text{OF weak-consis-drop a-ro} a-not-owns
  \text{acquired-all-acquired} \text{OF True ?take-sb O_{sb}}
  \text{all-acquired-append} \text{OF ?take-sb ?drop-sb}
  have a \in \text{read-only} S
  by auto
  with \text{share-all-until-volatile-write-thread-local} \text{OF ownership-distinct-ts}_{sb}
  \text{sharing-consis-ts}_{sb} j-bound jth j-owns]
  have a \in \text{read-only} \text{share} ?take-sb_j S_{sb}
      by (auto simp add: read-only-def S)
  hence a-dom: a \in \text{dom} \text{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_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 ∧ 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 o is-volatile-Write$S_{sb}$) sb_j)]
  have unshared-take: non-volatile-writes-unshared $S_{sb}$ (takeWhile (Not o is-volatile-Write$S_{sb}$) 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_j $S_{sb}$)
    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 \textit{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 \textit{thesis}  
by (simp add: buf-None)  
qed

hence m-a: m a = m_{sb} a  
by (simp add: m)

have tmps-commute: \( \delta_{sb}(t \mapsto (m_{sb} a)) = \( (\delta_{sb} | (\text{dom} \setminus \{t\}))(t \mapsto (m_{sb} a)) \) \)  
apply (rule ext)  
apply (auto simp add: restrict-map-def domIff)  
done

from suspend-nothing  
have suspend-nothing': \( \text{dropWhile} \ (\text{Not } \circ \text{volatile-Write}_{sb}) \ sb' = [] \)  
by (simp add: sb')

from \( \mathcal{D} \)  
have \( \mathcal{D}' : \mathcal{D}_{sb} = (\mathcal{D} \lor \text{outstanding-refs is-volatile-Write}_{sb} (sb@[\text{Read}_{sb} \text{volatile} a t (m_{sb} a)]) \neq \{\}) \)  
by (simp add: sb')

have \( (ts_{sb}', m_{sb}, S_{sb}') \sim (ts[i := (p_{sb}, i_{sb}'), \delta_{sb}(t \mapsto m a),()], \mathcal{D}, \text{acquired True } ?\text{take-sb} \) \( \mathcal{O}_{sb}, \text{release } ?\text{take-sb} \) \( \text{dom } S_{sb} \) \( \mathcal{R}_{sb} \) \( \delta_{sb}' \) \)  
apply (rule sim-config.intros)  
apply (simp add: m flush-commute ts_{sb}' \( \mathcal{O}_{sb}', \mathcal{R}_{sb}', \delta_{sb}', \mathcal{O}_{sb}', \mathcal{R}_{sb}', \delta_{sb}' \) )  
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' \( \mathcal{O}_{sb}', \mathcal{R}_{sb}', \delta_{sb}' \) )  
using leq  
apply (simp add: ts_{sb}')  
using i-bound i-bound' ts-sim ts-i True \( \mathcal{D}' \)  
apply (clarsimp simp add: Let-def nth-list-update  
outstanding-refs-conv m-a ts_{sb}' \( \mathcal{O}_{sb}', \mathcal{R}_{sb}', S_{sb}', \delta_{sb}', \mathcal{D}_{sb}' \) )  
split: if-split-asm )  
apply (rule tmps-commute)  
done

ultimately show \textit{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 ∈ 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'' sop'' A′ L′ R′ W′ 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''
  by simp

have (ts, m, S) ⇒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'∈ set sb
  by (subst sb-split) auto

from dropWhile-append1 [OF this, of (Not ◦ is-volatile-Write sb)]
have drop-app-comm:
  (dropWhile (Not ◦ 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 δ sb
  by (auto simp add: is sb)
then have tmps-commute:
  δ sb |' (dom δ sb − read-tmps sb'') =
    δ sb |' (dom δ sb − insert t (read-tmps sb''))
  apply −
  apply (rule ext)
  apply auto

477
from $D$

have $D': D_s = (D \lor \text{outstanding-refs is-volatile-Write}_s (sb@a[\text{Read}_s \text{volatile a} t (m_s a)]) \neq \{\})$
  by (auto simp: outstanding-refs-append)

have $(t_{s_b'}, m_{s_b'}, S_{s_b'}) \sim (t_s, m, S)$
  apply (rule sim-config-intros)
  apply (simp add: m flush-commute $t_{s_b'} O_{s_b'} R_{s_b'} \theta_{s_b'} sb'$)
  using (simp add: $S_{s_b'} t_{s_b'} sb' O_{s_b'} R_{s_b'} \theta_{s_b'}$)
  apply leq
  using $t_{s_b'}$ $O_{s_b'}$ $R_{s_b'}$ $\theta_{s_b'}$ $D_{s_b'}$
  apply (clarsimp simp add: Let-def nth-list-update is-sim drop-app-comm
    read-tmps-append suspends prog-instrs-append-Read$_{s_b}$ instrs-append-Read$_{s_b}$
    hd-prog-append-Read$_{s_b}$
    drop is$_{s_b'}$ $t_{s_b'} sb' O_{s_b'} R_{s_b'} \theta_{s_b'} D_{s_b'}$
    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'$'$
  $m_{s_b'}$ $S_{s_b'}$
  by (auto simp del: fun-upd-apply )
qed

next
  case (SBHWriteNonVolatile a D f A L R W)
  then obtain
    is$_{s_b'}$: is$_{s_b'} = \text{Write False a} (D, f) A L R W# is$_{s_b'}$ and
    $O_{s_b'}': O_{s_b'} = O_{s_b}$ and
    $R_{s_b'}': R_{s_b'} = R_{s_b}$ and
    $\theta_{s_b'}': \theta_{s_b'} = \theta_{s_b}$ and
    $D_{s_b'}': D_{s_b'} = D_{s_b}$ and
    sb': sb' = sb@[Write$_s$ False a (D, f) (f $\theta_{s_b}$) A L R W] and
    m$_{s_b'}$: m$_{s_b'}$ = m$_s$ and
    S$_{s_b'}$: S$_{s_b'}$ = S$_s$
  by auto

  from data-dependency-consistent-instrs [OF i-bound ts$_{s_b}$-i]
  have D-tmps: D $\subseteq$ dom $\theta_{s_b}$
  by (simp add: is$_{s_b}$)

  from safe-memop-flush-sb [simplified is$_{s_b}$]
obtain a-owned': a ∈ acquired True sb O_{sb} and a-unshared': a /∈ dom (share ?drop-sb S) and

rels-cond: ∀ j < length ts. i ≠ j → a /∈ dom (released (ts!j))

by cases auto

from a-owned' acquired-all-acquired
have a-owned'': a ∈ O_{sb} ∪ all-acquired sb
by auto

{ 
fix j
fix p_{j} is_{j} R_{j} D_{j} \theta_{j} sb_{j}
assume j-bound: j < length ts_{sb}
assume ts_{sb}-j: ts_{sb}!j = (p_{j}, is_{j}, \theta_{j}, sb_{j}, D_{j}, O_{j}, R_{j})
assume neq-i-j: i \neq j
have a /∈ O_{j} ∪ all-acquired sb_{j}
proof –
  from ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i ts_{sb}-j] a-owned''
  show ?thesis
    by auto
qed
} note a-unowned-others = this

have a-unshared: a /∈ dom (share sb S_{sb})
proof
assume a-share: a ∈ 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-owned'' a-share]
have a ∈ dom (share ?drop-sb S)
  by (simp add: S)
with a-unshared'
show False
  by auto
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 –
  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.
  with a-owned'

479
have non-volatile-owned-or-read-only False $S_{sb} O_{sb}$ (sb @ [Write_{sb} False a (D,f) (f $\vartheta_{sb}$) A L R W])
  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 $\theta$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 $\vartheta_{sb}$) A L R W])
    $\subseteq$
    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 $\theta$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 o is-volatile-Write_{sb})
    (sb @ [Write_{sb} False a (D,f) (f $\vartheta_{sb}$) A L R W]) $O_{sb}$)
    (dropWhile (Not o is-volatile-Write_{sb}) (sb @ [Write_{sb} False a (D,f) (f $\vartheta_{sb}$) A L R W]))
    $\subseteq$
    read-only-reads (acquired True (takeWhile (Not o is-volatile-Write_{sb}) sb) $O_{sb}$)
    (dropWhile (Not o 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 $O_{sb}$ $\cup$ all-acquired (sb @ [Write_{sb} False a (D,f) (f $\vartheta_{sb}$) A L R W]) $\subseteq$ $O_{sb}$ $\cup$
    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 $\theta$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 $\theta$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] have history-consistent \( \theta \sb \) (hd-prog \( p \sb \) sb) sb.
with valid-write-sops [OF i-bound ts\sb] D-tmps valid-implies-valid-prog-hd [OF i-bound ts\sb valid] have history-consistent \( \theta \sb \) (hd-prog \( p \sb \) sb (sb@ [Write\sb False a (D,f) (f \( \theta \sb \)) A L R W]))
apply (rule history-consistent-appendI)
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′ O\sb′ \( \theta \sb \)′)
qed

have valid-reads': valid-reads m\sb ts\sb′
proof
from valid-reads [OF i-bound ts\sb] have reads-consistent False \( \theta \sb \) m\sb sb.
from reads-consistent-snoc-Write\sb [OF this] have reads-consistent False \( \theta \sb \) m\sb (sb@ [Write\sb False a (D,f) (f \( \theta \sb \)) A L R W]).
from valid-reads-nth-update [OF i-bound this] show ?thesis by (simp add: ts\sb′ is\sb′ O\sb′ \( \theta \sb \)′)
qed

have valid-sharing': valid-sharing \( S \sb \) ts\sb′
proof (intro-locales)
from outstanding-non-volatile-writes-unshared [OF i-bound ts\sb] a-unshared have non-volatile-writes-unshared \( S \sb \)
apply (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′
apply (auto simp add: outstanding-non-volatile-writes-unshared-nth-update)
by (simp add: ts\sb′ is\sb′ O\sb′ \( \theta \sb \)′ \( S \sb \)′)

next
from sharing-consis [OF i-bound ts\sb] have sharing-consistent \( S \sb \) \( O \sb \) sb.
then have sharing-consistent \( S \sb \) \( O \sb \) \( p \sb \) sb \( \theta \sb \)
apply (auto simp add: sharing-consistent-append)
from sharing-consis-nth-update [OF i-bound this] show sharing-consis \( S \sb \) ts\sb′
apply (auto simp add: sharing-consis-nth-update)
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]] show read-only-unowned \( S \sb \) ts\sb'
apply (auto simp add: read-only-unowned-nth-update)

next
from unowned-shared-nth-update [OF i-bound ts\sb subset-refl]

481
show unowned-shared $S_{sb}'ts_{sb}'$
  by (simp add: $ts_{sb}'is_{sb}sb'O_{sb}'\partial_{sb}'S_{sb}'$)
next
from a-unshared
have a $\notin$ read-only (share sb $S_{ab}$)
  by (auto simp add: read-only-def dom-def)
with no-outstanding-write-to-read-only-memory [OF i-bound ts_{sb}!]
have no-write-to-read-only-memory $S_{sb} (sb @ [Write_{sb} \text{ False} a (D,f) (f \partial_{sb}) A L R W])$
  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-tmp-distinct [OF i-bound ts_{sb}!]
have distinct-load-tmps is_{sb}'
  by (auto split: instr.splits simp add: is_{sb})
from load-tmp-distinct-nth-update [OF i-bound this]
show load-tmps-distinct ts_{sb}'
  by (simp add: $ts_{sb}'is_{sb}sb'O_{sb}'\partial_{sb}'$)
next
from load-tmps-distinct [OF i-bound ts_{sb}!]
have distinct-read-tmps sb.
  hence distinct-read-tmps (sb @ [Write_{sb} \text{ False} a (D,f) (f \partial_{sb}) A L R W])
  by (clarsimp simp add: read-tmps-append is_{sb})
from load-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct ts_{sb}'
  by (simp add: $ts_{sb}'is_{sb}sb'O_{sb}'\partial_{sb}'$)
next
from load-tmps-read-tmps-distinct [OF i-bound ts_{sb}!]
  load-tmps-distinct [OF i-bound ts_{sb}!]
have $is_{sb}' \cap$ read-tmps (sb @ [Write_{sb} \text{ False} a (D,f) (f \partial_{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: $ts_{sb}'is_{sb}sb'O_{sb}'\partial_{sb}'$)
qed

have valid-sops': valid-sops ts_{sb}'
proof –
from valid-store-sops [OF i-bound ts_{sb}!]
obtain valid-Df: valid-sop (D,f) and
  valid-store-sops': $\forall$ sop$\in$store-sops is_{sb}' . valid-sop sop
  by (auto simp add: is_{sb})
from valid-Df valid-write-sops [OF i-bound ts_{sb}!]
have valid-write-sops': $\forall$ sop$\in$write-sops (sb@ [Write_{sb} \text{ False} a (D,f) (f \partial_{sb}) A L R W]).
482
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 O sb θ 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 False a (D f) (f θ 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 O sb θ 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: ts sb sb O 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@[Write sb False a (D f) (f θ 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

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@[Write sb False a (D f) (f θ sb) A L R W])

483
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 (sb @ [Write \sb False a (D f) (f \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 ?thesis
by (simp add: ts\sb′ sb′)
qed

from valid-store-sops [OF i-bound ts\sb-i, rule-format]
have valid-sop (D,f) by (auto simp add: is\sb)
then interpret valid-sop (D,f).
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 = Write False 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, Write False a (D,f) A L R W# is\sb′, 
\sb, ().
D, acquired True ?take-sb \sb, release ?take-sb (dom (S\sb)) \sb)
by simp

from direct-memop-step.WriteNonVolatile [OF ]
have (Write False a (D, f) A L R W# is\sb′, 
\sb, (),m,D,acquired True ?take-sb \sb, release ?take-sb (dom (S\sb)) \sb, S) \rightarrow 
(is\sb′, 
\sb, (), m(a := f \sb), D, acquired True ?take-sb \sb, 
release ?take-sb (dom (S\sb)) \sb).
from direct-computation.concurrent-step.Memop [OF i-bound′ ts-i this]
have (ts, m, S) \Rightarrow (ts\i := (p\sb, is\sb′, \sb, ().
D, acquired True ?take-sb \sb, 
release ?take-sb (dom (S\sb)) \sb), m(a := f \sb),S),
moreover

484
have \( \forall j < \text{length } ts_{sb} . \ i \neq j \rightarrow \)
\( (\text{let } (-,\cdots,-,sb_j,\cdots,-) = ts_{sb} ! j) \)
in a \( \notin \text{outstanding-refs } is\text{-non-volatile-Write}_{sb} \) (takeWhile (Not \circ is\text{-volatile-Write}_{sb}) sb_j))
proof
{\ 
  fix j p_j is_j O_j R_j D_j jth \ acq_j x_{s_j} s_{b_j} \
  assume j-bound: \ j < \text{length } ts_{sb} \
  assume neq-i-j: i \neq j \
  assume jth: ts_{sb} ! j = (p_j, \text{is}_j, x_{s_j}, s_{b_j}, D_j, O_j, R_j) 
}
have a \( \notin \text{outstanding-refs } is\text{-non-volatile-Write}_{sb} \) (takeWhile (Not \circ is\text{-volatile-Write}_{sb}) sb_j)

  proof
    assume a-in: a \( \in \text{outstanding-refs } is\text{-non-volatile-Write}_{sb} \) (takeWhile (Not \circ is\text{-volatile-Write}_{sb}) sb_j) 
    hence a \( \in \text{outstanding-refs } is\text{-non-volatile-Write}_{sb} \) sb_j 
    using outstanding-refs-append [of is\text{-non-volatile-Write}_{sb} (takeWhile (Not \circ is\text{-volatile-Write}_{sb}) sb_j)]
    (dropWhile (Not \circ is\text{-volatile-Write}_{sb}) sb_j)]
    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 jth]]
    have j-owns: a \( \in O_j \cup \text{all-acquired } s_{b_j} \)
    by auto 
    from j-owns a-ownedownership-distinct [OF i-bound j-bound neq-i-j ts_{sb} \(-i jth\)]
    show False
    by auto 
  qed
}
thus ?thesis by (fastforce simp add: Let-def)
qed

note flush-commute = flush-all-until-volatile-write-append-non-volatile-write-commute 
[OF True i-bound ts_{sb} \(-i this\)]

from suspend-nothing
have suspend-nothing': (dropWhile (Not \circ is\text{-volatile-Write}_{sb}) sb') = []
  by (simp add: sb')

from D
have D': D_{sb} = (D \lor \text{outstanding-refs } is\text{-volatile-Write}_{sb} 
(s_{b}@[Write sb False a (D,f) (f s_{sb}) 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, \text{acquired } \text{?take-sb } O_{sb}, 
    \text{release } \text{?take-sb } (\text{dom } (S_{sb})) (R_{sb}))],
   m(a:=f \vartheta_{sb},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-Write-commute
[OF i-bound ts$_{sb}$-i [simplified is$_{sb}$]]
apply (clarsimp simp add: $S_{sb}'$ $ts_{sb}'$ $sb'$ $O_{sb}'$ $R_{sb}'$ $\theta_{sb}'$ $D_{sb}'$)
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$_{sb}'$ $O_{sb}'$ $R_{sb}'$ $\theta_{sb}'$ $D_{sb}'$
    suspend-nothing' flush-all
    acquired-append release-append split: 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

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 $\circ$ is-volatile-Write$_{sb}$), simplified, OF volatile-r]
    obtain a' v' sb'' sop' A' L' R' W' where
        sb-split: sb = takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb @ Write$_{sb}$ True a' sop' v' A' L'
        R' W'# sb''
        and
        drop: dropWhile (Not $\circ$ 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-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]

from \mathcal{D}
have \mathcal{D}': \mathcal{D}_{\sb{}} = (\mathcal{D} \vee \ outstanding-ref\s\ is-volatile-Write\sb{} (sb@[Write\sb{} False a (D,f) (f \ \varnothing_{\sb{}}) A L R W]) \neq \{\})
by (auto simp: outstanding-ref\s\ 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{}' \varnothing_{\sb{'}} sb')
using share-all-until-volatile-write-Write-commute
[OF i-bound ts\sb{}-i [simplified is\sb{}]]
apply (clarsimp simp add: S S\sb{}' ts\sb{}' sb' O\sb{}' R\sb{}' \varnothing_{\sb{'}} sb')
using leq
apply (simp add: ts\sb{}')
using i-bound i-bound' ts-sim ts-i is-sim
apply (clarsimp simp add: Let-def nth-list-update is-sim drop is\sb{} ts\sb{} sb' O\sb{}' R\sb{}' S\sb{}' \varnothing_{\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-flushs'
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
  \varnothing_{\sb{'}}: \varnothing_{\sb{'}} = \varnothing_{\sb{}} and
  D\sb{}': D\sb{} = True and
  sb': sb' = sb@[Write\sb{} True a (D, f) (f \varnothing_{\sb{}}) A L R W] and
  m\sb{}': m\sb{} = m\sb{}' and
  S\sb{}': S\sb{} = S\sb{}
by auto

  from data-dependency-consistent-instrs [OF i-bound ts\sb{}-i]
  have D-subset: D \subseteq dom \varnothing_{\sb{}}
by (simp add: is\sb{})

487
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: \( A \subseteq \text{dom} (\text{share } ?\text{drop-sb } S) \cup \text{acquired } \text{True } sb \ O_{sb} \) and
R-acq: \( R \subseteq \text{acquired } \text{True } sb \ O_{sb} \) 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': \( a \notin \) read-only (share ?\text{drop-sb } S)
by cases auto

from a-unowned-others-ts ts-sim leq
have a-unowned-others:
\[ \forall j < \text{length } ts_{sb}. i \neq j \rightarrow \\
(\text{let } (-,-,-,sb_j,\cdot,\cdot,\cdot) = ts_{sb}!j \text{ in } \\
a \notin \text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_j) O_j \land \\
a \notin \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_j)) 
\]
apply (clarsimp simp add: Let-def)
subgoal for j
apply (drule-tac x=j in spec)
apply (auto simp add: dom-release-takeWhile)
done
done

have a-not-ro: \( a \notin \) read-only (share sb \( S_{sb} \))
proof
assume a: \( a \in \) read-only (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}
\( \langle \text{read-only-unowned } S_{sb} \ ts_{sb} \rangle \ i\text{-bound } ts_{sb}-i \) a-unowned-others a]
have a \( \in \) read-only (share ?\text{drop-sb } S)
by (simp add: S)
with a-not-ro' show False by simp
qed

from A-unowned-by-others-ts ts-sim leq
have A-unowned-by-others:
\[ \forall j < \text{length } ts_{sb}. i \neq j \rightarrow \\
(\text{let } (-,-,-,sb_j,\cdot,\cdot,\cdot) = ts_{sb}!j \text{ in } \\
A \cap (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_j) O_j \cup \\
\text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb_j)) = \{\}) 
\]
apply (clarsimp simp add: Let-def)
subgoal for j
apply (drule-tac x=j in spec)
apply (force simp add: dom-release-takeWhile)

488
\begin{itemize}
\item \textbf{have} a-not-acquired-others: \( \forall j < \text{length} \ (\text{map} \ O\sb j \ \text{ts}_{sb}) \). \( i \neq j \rightarrow \)
\[ (\text{let} \ (O_j, sb_j) = (\text{map} \ O\sb sb \ \text{ts}_{sb})|j \text{ in a \notin \ all-acquired sb}_j) \]
\item \textbf{proof} --
\end{itemize}

\begin{verbatim}
{  
  fix j O_j sb_j
  assume j-bound: j < \text{length} \ (\text{map} \ \text{owned} \ \text{ts}_{sb})
  assume neq-i-j: i \neq j
  assume ts_{sb}-j: (\text{map} \ O\sb sb \ \text{ts}_{sb})|j = (O_j, sb_j)
  assume conflict: a \in \ all-acquired sb_j
  have False
  proof --
  from j-bound leq  
  have j-bound': j < \text{length} \ \text{ts}_{sb}
    by auto
  from j-bound have j-bound'': j < \text{length} \ \text{ts}_{sb}
    by auto
  from j-bound' have j-bound'''': j < \text{length} \ \text{ts}
    by simp
  let ?take-sb_j = (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}_j)
  let ?drop-sb_j = (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}_j)
  from ts-sim [\text{rule-format}, \text{OF} \ j-bound'''] \ \text{ts}_{sb}-j \ j-bound'''
    obtain p_j suspends_j is_{sbj} \ R_j \ D_{sbj} \ D_j \ \hat{\theta}_{sbj} \ \text{where}
      ts_{sb}-j \ : \ \text{ts}_{sb}!j = (p_j, is_{sbj}, \ \hat{\theta}_{sbj}, \ sb_j, D_{sbj}, O_j, R_j) \ \text{and}
      suspends_j: \ \text{suspends}_{sbj} = \text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}_j \ \text{and}
      is_j: \ \text{instrs} \ \text{suspends}_{sbj} \ @ \ \text{is}_{sbj} = \text{is}_j \ @ \ \text{prog-instrs} \ \text{suspends}_{sbj} \ \text{and}
      D_j: D_{sbj} = (D_j \ \vee \ \text{outstanding-refs} \ \text{is-volatile-Write}_{sb} \ sb_j \neq \{\}) \ \text{and}
      ts_j: \ \text{ts}_j|j = (\text{hd-prog} \ p_j \ \text{suspends}_{sbj}, \ is_j, \ \hat{\theta}_{sbj} | \ (\text{dom} \ \hat{\theta}_{sbj} - \text{read-tmps} \ \text{suspends}_{sbj}),(),)
      \ \text{and}
      D_j, \ acquired \ True \ ?take-sb_j \ O_j,
      release \ ?take-sb_j \ (\text{dom} \ \mathcal{S}_{sb}) \ R_j)
    apply (cases \ ts_{sb}|j)
    apply (force simp add: Let-def)
  done

  from \ \text{a-unowned-others} \ [\text{rule-format,OF} \ - \ \text{neq-i-j}] \ \text{ts}_{sb}-j \ j-bound
    obtain \ \text{a-unacq}: \ a \ \notin \ \text{acquired} \ True \ ?take-sb_j \ O_j \ \text{and} \ \text{a-not-shared}: \ a \ \notin \ \text{all-shared} \ ?take-sb_j
      by auto
      have conflict-drop: a \in \ all-acquired \ suspends_j
      proof (rule ccontr)
        assume a \notin \ all-acquired \ suspends_j
        with \ \text{all-acquired-append} \ [\text{of} \ ?take-sb_j \ ?drop-sb_j] \ \text{conflict}
        have a \in \ all-acquired \ ?take-sb_j
    done

  489
\end{verbatim}
by (auto simp add: suspends) 
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 \( \exists \text{sop a'} \, \forall \theta \in \text{A} \) 
  suspends\(_j\) = ys @ Write\(_sb\) True a' \text{sop } \theta \in \text{A} \cap \text{R} \cap \text{W} 
  then 
  obtain a' \text{sop } \theta \in \text{A} \cap \text{R} \cap \text{W} [where] 
split-suspends\(_j\): suspends\(_j\) = ys @ Write\(_sb\) True a' \text{sop } \theta \in \text{A} \cap \text{R} \cap \text{W} 
  (is suspends\(_j\) = ?suspends) and 
a-A': a \in \text{A} 
  by blast 

  from sharing-consis [OF j-bound"" ts\(_{sb-j}\)] 
  have sharing-consis-j: sharing-consistent \(\mathcal{S}_{sb} \text{O}_j \text{sb}_j\). 
  then have A'-R': A' \cap R' = {} 
  by (simp add: sharing-consistent-append [of - ?take-sb_j ?drop-sb_j, simplified] 
  suspends\(_j\) [symmetric] split-suspends\(_j\) sharing-consistent-append) 
  from valid-program-history [OF j-bound"" ts\(_{sb-j}\)] 
  have causal-program-history is\(_{sbj s_bj}\). 
  then have cph: causal-program-history is\(_{sbj s_bj}\) ?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\(_j\), 
  \(\theta\)\(_sbj\) \mid (\text{dom} \(\theta\)\(_sbj\) - \text{read tmtps suspends}\(_j\)\(),(), 
  \(\mathcal{D}_j\), acquired True ?take-sb\(_j\) \text{O}_j, release ?take-sb\(_j\) (\text{dom} \(\mathcal{S}_{sb}\) \text{R}_j) 
  by auto 
  from valid-last-prog [OF j-bound"" ts\(_{sb-j}\)] have last-prog: last-prog p\(_j\) sb\(_j\) = p\(_j\). 
  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’’ ts_ab-j]
have reads-consis-j: reads-consistent False \( S_j \) \( m_{sb} \) sb.

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing \( S_{sb} \) \( t_{sb} \) j-bound’’ \( t_{sb-j} \) 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’’ \( t_{sb-i} \) \( t_{sb-j} \)]
have outstanding-refs is-Write\( sb \) \( ?\text{drop-sb} \cap \) outstanding-refs is-non-volatile-Read\( sb \) suspends\( j \) = {}
by (simp add: \( m \) 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 \( ?\text{drop-sb} \) \( m \)) \( ys@\) [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_ab-j]
have \( \forall \text{sop} \in \text{write-sops} \) \( (?\text{take-sb}_j@?\text{suspends}) \). valid-sop \( \text{sop} \)
by (simp add: split-suspends\( j \) [symmetric] suspends\( j \))
then obtain valid-sops-take: \( \forall \text{sop} \in \text{write-sops} \) \( ?\text{take-sb}_j \). valid-sop \( \text{sop} \) and valid-sops-drop: \( \forall \text{sop} \in \text{write-sops} \) \( ys@\) [Write\( sb \) True \( a' \) sop' \( v' \) A' L' R' W']. valid-sop \( \text{sop} \)
apply (simp only: write-sops-append)
apply auto
done

from read-tmps-distinct [OF j-bound’’ ts_ab-j]
have distinct-read-tmps \( ?\text{take-sb}_j@?\text{suspends}_j \)
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-drop 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_ab-j]
have h-consis:
history-consistent \( \theta_{sb} \) (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

491
have last-prog-hd-prog: last-prog (hd-prog p j sbj) ?take-sbj = (hd-prog p j suspendsj)
proof –
from last-prog have last-prog p j (?take-sbj@?drop-sb) = p j
by simp
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog p j suspendsj) ?take-sbj = (hd-prog p j suspendsj)
by (simp only: split-suspendsj [symmetric] suspendsj)
moreover
have last-prog (hd-prog p j (?take-sbj @ suspendsj)) ?take-sbj = last-prog (hd-prog p j suspendsj) ?take-sbj
apply (simp only: split-suspendsj [symmetric] suspendsj)
by (rule last-prog-hd-prog-append)
ultimately show ?thesis
by (simp add: split-suspendsj [symmetric] suspendsj)
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 suspendsj) suspendsj
by (simp add: split-suspendsj [symmetric] suspendsj)
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 suspendsj [symmetric] split-suspendsj)

have acq-simp:
acquired True (ys @[Write sb True a' sop' v' A' L' R' W'])
(acquired True ?take-sb O j) =
acquired True ys (acquired True ?take-sb 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' isj [simplified split-suspendsj] cph [simplified suspendsj]
hist-consis' [simplified split-suspendsj] valid-sops-drop
distinct-read-tmps-drop [simplified split-suspendsj]
no-volatile-Read sb-volatile-reads-consistent [OF no-vol-read], where
S=share ?drop-sb S]
obtain isj' R j' where
isj': instrs zs @ is sbj = isj' @ prog-instrs zs and
steps-ys: (?ts', flush ?drop-sb m, share ?drop-sb 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',
Θ sbj |' (dom θ sbj − read-tmps zs),

492
(), True, acquired True ys (acquired True ?take-sb \(O_j\)) \(\cup A' - R', R'_j\)],

- flush \((ys @ [Write^{sb}_{sb} True a' sop' v' A' L' R' W'])\) (flush ?drop-sb m),
- share \((ys @ [Write^{sb}_{sb} True a' sop' v' A' L' R' W'])\) (share ?drop-sb \(S\))

\((is \ (-, -, -) \Rightarrow d^* (?ts-ys, ?m-ys, ?shared-ys))\)

\(\text{by (auto simp add: acquired-append outstanding-refs-append)}\)

\(\text{from i-bound'} \text{ have i-bound-ys: } i < \text{ length ?ts-ys by auto}\)

\(\text{from i-bound'} \text{ neq-i-j have ts-ys-i: } \text{ ?ts-ys!i} = (p^{sb}_{sb}, i^{sb}_{sb}, \emptyset^{sb}_{sb}).\)

\(D^{sb}_{sb}, \text{ acquired True sb } O^{sb}_{sb}, \text{ release sb (dom } S^{sb}_{sb}) R^{sb}_{sb}\)

\(\text{by simp note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]}\)

\(\text{from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]}\)

\(\text{have safe-delayed (?ts-ys, ?m-ys, ?shared-ys).}\)

\(\text{from safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is^{sb}_{sb}]}\)

\(\text{have a-unowned: }\)

\(\forall j < \text{ length ?ts-ys. } i \neq j \rightarrow (\text{let } (O_j) = \text{ map owned ?ts-ys!j in a } \notin O_j)\)

\(\text{apply cases}\)

\(\text{apply (auto simp add: Let-def is^{sb}_{sb})}\)

\(\text{done}\)

\(\text{from a-A'} a\text{-unowned [rule-format, of j] neq-i-j j-bound'} A' - R'\)

\(\text{show False by (auto simp add: Let-def)}\)

\(\text{next}\)

\(\text{assume } \exists A L R W ys zs. \text{ suspends}_j = ys @ \text{ Ghost}^{sb}_{sb} A L R W# zs \land a \in A\)

\(\text{then}\)

\(\text{obtain } A' L' R' W' ys zs \text{ where}\)

\(\text{split-suspends}: \text{ suspends}_j = ys @ \text{ Ghost}^{sb}_{sb} A' L' R' W'# zs\)

\(\text{is suspends}_j = ?suspends \text{ and}\)

\(a - A': a \in A'\)

\(\text{by blast}\)

\(\text{from sharing-consis [OF j-bound'' ts^{sb}_{sb}-j]}\)

\(\text{have sharing-consis-j: sharing-consistent } S^{sb}_{sb} O_j sb_j.\)

\(\text{then have A'R': } A' \cap R' = \{\}\)

\(\text{by (simp add: sharing-consistent-append [of - - ?take-sb_j ?drop-sb_j, simplified] suspends_j [symmetric] split-suspends_j sharing-consistent-append)}\)

\(\text{from valid-program-history [OF j-bound'' ts^{sb}_{sb}-j]}\)

\(\text{have causal-program-history is^{sb}_{sb} sb_j.}\)

\(\text{then have cph: causal-program-history is^{sb}_{sb} ?suspends}\)

\(\text{apply} –\)

\(\text{apply (rule causal-program-history-suffix [where sb=?take-sb_j])}\)

\(\text{apply (simp only: split-suspends_j [symmetric] suspends_j)}\)

\(\text{apply (simp add: split-suspends_j)}\)

\(\text{done}\)
from ts\_i \text{ neq-i-j j-bound}

\begin{align*}
& \text{have ts}'\_j: ?ts\_i'^j = (\text{hd-prog } p\_j \text{ suspends}_j, i\_j, \\
& \text{dom } \theta_{sb} \mid (\text{dom } \theta_{sb} - \text{read-tmps suspends}_j),(), \\
& D\_j, \text{acquired True } ?\text{take-sb}_j \ O\_j, \text{release } ?\text{take-sb}_j (\text{dom } S\_sb) \ R\_j \\
& \text{by auto} \\
& \text{from valid-last-prog } [\text{OF } j\text{-bound'' ts}_{sb}\_j] \textbf{have last-prog: last-prog } p\_j \text{ sb}_j = p\_j. \\
& \text{then} \\
& \text{have lp: last-prog } p\_j \text{ suspends}_j = p\_j \\
& \text{apply} - \\
& \text{apply } (\text{rule last-prog-same-append } \text{where } sb=?\text{take-sb}_j) \\
& \text{apply } (\text{simp only: split-suspends}_j [\text{symmetric} ] \text{ suspends}_j) \\
& \text{apply simp} \\
& \text{done} \\
\end{align*}

\begin{align*}
& \text{from valid-reads } [\text{OF } j\text{-bound'' ts}_{sb}\_j] \\
& \text{have reads-consis-j: reads-consistent False } O\_j \text{ m}_{sb}, sb\_j. \\
& \text{from reads-consistent-flush-all-untl-volatile-write } [\text{OF } \text{valid-ownership-and-sharing} S\_sb \text{ ts}_{sb}\_j] \\
& j\text{-bound'' ts}_{sb}\_j \text{ this]} \\
& \text{have reads-consis-m-j: reads-consistent True (acquired True } ?\text{take-sb}_j \ O\_j) \text{ m suspends}_j \\
& \text{by } (\text{simp add: m suspends}_j) \\
& \text{from outstanding-non-write-non-vol-reads-drop-disj } [\text{OF i-bound } j\text{-bound'' neq-i-j ts}_{sb}\_i \text{ ts}_{sb}\_j] \\
& \text{have outstanding.refs is-Write}_{sb} \cap \text{outstanding.refs is-non-volatile-Read}_{sb} \\
& \text{suspends}_j = \{\} \\
& \text{by } (\text{simp add: suspends}_j) \\
& \text{from reads-consistent-flush-independent } [\text{OF this reads-consis-m-j} \\
& \text{have reads-consis-flush-suspend: reads-consistent True (acquired True } ?\text{take-sb}_j \ O\_j) \\
& (\text{flush } ?\text{drop-sb m}) \text{ suspends}_j. \\
& \text{hence reads-consis-ys: reads-consistent True (acquired True } ?\text{take-sb}_j \ O\_j) \\
& (\text{flush } ?\text{drop-sb m}) (ys@[\text{Ghost}_{sb} A' L' R' W']) \\
& \text{by } (\text{simp add: split-suspend}s\_j \text{ reads-consistent-append}) \\
& \text{from valid-write-sops } [\text{OF j-bound'' ts}_{sb}\_j] \\
& \text{have } \forall \text{sop } \in \text{write-sops } (?\text{take-sb}_j@?\text{suspends}). \text{valid-sop sop} \\
& \text{by } (\text{simp add: split-suspends}_j [\text{symmetric} ] \text{ suspends}_j) \\
& \text{then obtain } \text{valid-sops-take: } \forall \text{sop } \in \text{write-sops } ?\text{take-sb}_j. \text{valid-sop sop} \ \textbf{and} \\
& \text{valid-sops-drop: } \forall \text{sop } \in \text{write-sops } (ys@[\text{Ghost}_{sb} A' L' R' W']). \text{valid-sop sop} \\
& \text{apply } (\text{simp only: write-sops-append}) \\
& \text{apply auto} \\
& \text{done} \\
& \text{from read-tmps-distinct } [\text{OF j-bound'' ts}_{sb}\_j] \\
& \text{have distinct-read-tmps } (?\text{take-sb}_j@?\text{suspends}_j) \\
& \text{by } (\text{simp add: split-suspend}s\_j [\text{symmetric} ] \text{ suspends}_j) \\
& \text{then obtain} 
\end{align*}
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 ?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 s_b_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 ?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-j]
have no-vol-read: outstanding-refs is-volatile-Read_{sb}
(y_s@[Ghost_{sb} A’ L’ R’ W’]) = {}
by (auto simp add: outstanding-refs-append suspends_j [symmetric]
split-suspends_j)

have acq-simp:
acquired True (y_s@[Ghost_{sb} A’ L’ R’ W’])
(acquired True ?take-sb_j O_j) =
acquired True y_s (acquired True ?take-sb_j O_j) ∪ A’ − R’
by (simp add: acquired-append)

from flush-store-buffer-append [where sb=y_s@[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-y_s

495
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]

\textbf{obtain} is_{j'} / R_{j'} \textbf{ where}

is_{j'}: instrs zs @ is_{sbj} = is_{j'} @ prog-instrs zs \textbf{ and}
steps-ys: (?ts', flush ?drop-sb m, share ?drop-sb S) \Rightarrow d^

(?ts'[j]:=(last-prog

(hd-prog p_j (Ghost_{sb} A' L' R' W'# zs)) (ys@[Ghost_{sb} A' L' R' W']),
is_{j'},
\emptyset_{sbj} |' (dom \emptyset_{sbj} \cdot \text{read-tmps zs}),
(),
D_j \lor \text{outstanding-refs is-volatile-Write}_{sb} (ys @[Ghost_{sb} A' L' R'
W']) \neq \{\}, \text{acquired 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^

(by (auto simp add: acquired-append)

\textbf{from} i-bound' \textbf{ have} i-bound-ys: i < length ?ts-ys
\textbf{by} auto

\textbf{from} i-bound' neq-i-j
\textbf{ have} ts-ys-i: ?ts-ys!i = (p_{sb}, is_{sb}, \emptyset_{sb},(),
D_{sb}, \text{acquired True sb O_{sb}, release sb (dom S_{sb}) R_{sb})
\textbf{by} simp
\textbf{ note} conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

\textbf{from} safe-reach-safe-rtrancl [OF safe-reach conflict-computation]
\textbf{ have} safe-delayed (?ts-ys,?m-ys,?shared-ys).

\textbf{from} safe-delayedE [OF this i-bound-ys ts-ys-i, simplified is_{sb}]
\textbf{ have} a-unowned:
\forall j < length ?ts-ys. i\neq j \rightarrow (let (O_j) = \text{map owned ?ts-ys!j in a} \notin O_j)
\textbf{apply} cases
\textbf{apply} (auto simp add: Let-def is_{sb})
\textbf{done}
\textbf{from} a-A' a-unowned [rule-format, of j neq-i-j j-bound' A'-R'
\textbf{show} False
\textbf{by} (auto simp add: Let-def)
\textbf{qed}
\textbf{qed}

\textbf{thus} ?thesis
\textbf{by} (auto simp add: Let-def)
\textbf{qed}

\textbf{have} A-unused-by-others:
\forall j<length (map O_{sb} ts_{sb}). i \neq j \rightarrow

496
\[
\text{let } (O_j, \text{sb}_j) = \text{map } O-\text{sb} \text{ ts}_j! j
\]
\[
\text{in } A \cap \text{outstanding-refs is-volatile-Write}_a \text{ sb}_j = \{\}
\]

**proof** –

\[
\{ \text{fix } j \text{ O}_j \text{ sb}_j \\
\text{assume j-bound: } j < \text{length (map owned ts}_j) \\
\text{assume neq-i-j: } i \neq j \\
\text{assume ts}_j-j: \text{ (map } O-\text{sb ts}_j)!j = (O_j, \text{sb}_j) \\
\text{assume conflict: } A \cap \text{outstanding-refs is-volatile-Write}_a \text{ sb}_j \neq \{\}
\]

**have** False

**proof** –

\[
\text{from j-bound leq} \\
\text{have j-bound'}: j < \text{length ts}_j \\
\text{by auto} \\
\text{from j-bound have j-bound''}: j < \text{length ts}_j \\
\text{by auto} \\
\text{from j-bound' have j-bound''': j < \text{length ts}_j} \\
\text{by simp}
\]

**from** conflict **obtain** a' **where**

a'-in: a' ∈ A and

a'-in-j: a' ∈ outstanding-refs is-volatile-Write_a sb_j

by auto

\[
\text{let } ?\text{take-sb}_j = (\text{takeWhile (Not o is-volatile-Write}_a \text{ sb}_j) \\
\text{let } ?\text{drop-sb}_j = (\text{dropWhile (Not o is-volatile-Write}_a \text{ sb}_j)
\]

**from** ts-sim [rule-format, OF j-bound'' ts_j-j j-bound'']

**obtain** p_j suspends_j is_j \text{sb}_j D_j R_j \text{sbj} \text{isj} \text{where}

\[
t_s{s}_j: ts_j! j = (p_j, is_j, \text{sbj}, D_j, \text{sbj} O_j, R_j) \text{ and} \\
\text{suspends}_j: \text{suspends}_j = ?\text{drop-sb}_j \text{ and} \\
\text{isj: instrs suspends}_j @ is_j = is_j @ \text{prog-instrs suspends}_j \text{ and} \\
\text{D}_j: D_j = (D_j \checkmark \text{outstanding-refs is-volatile-Write}_a \text{ sb}_j \neq \{\}) \text{ and} \\
\text{ts}_j: tsj! = (\text{hd-prog p}_j \text{ suspends}_j, \text{isj},  \\
\text{isj} | \text{dom isj} - \text{read-tmps suspends}_j)(,), D_j,  \\
\text{ acquired True ?take-sb}_j O_j,  \\
\text{ release ?take-sb}_j (\text{dom S}_j) R_j)
\]

**apply** (cases ts_j-j)

**apply** (force simp add: Let-def)

**done**

**have** a' ∈ outstanding-refs is-volatile-Write_a suspends_j

**proof** –

\[
\text{from a'-in-j} \\
\text{have a' ∈ outstanding-refs is-volatile-Write}_a (?\text{take-sb}_j @ ?\text{drop-sb}_j)
\]

by simp

**thus** ?thesis

**apply** (simp only: outstanding-refs-append suspends_j)

**apply** (auto simp add: outstanding-refs-conv dest: set-takeWhileD)

**done**

497
qed

from split-volatile-Write\sb-in-outstanding-refs [OF this]

obtain sop v ys zs A’ L’ R’ W’ where
  split-suspends\sb: suspends\sb = ys @ Write\sb True a’ sop v A’ L’ R’ W’# zs (is suspends\sb = ?suspends)
  by blast

from direct-memop-step.WriteVolatile [where \varnothing =\varnothing\sb and m = flush ?drop-sb m]
have (Write True a (D, f) A L R W# is\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) \rightarrow
  (is\sb’, \varnothing\sb, (, (flush ?drop-sb m)(a := f \varnothing\sb), True, acquired True sb O\sb \cup
  A – R, Map.empty;
  share ?drop-sb S @\sb R \subseteq 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’, \varnothing\sb’, (, True, acquired True sb O\sb \cup A – R, Map.empty)),
  (flush ?drop-sb m)(a := f \varnothing\sb), share ?drop-sb S @\sb W R @\sb 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\sb])
  apply (simp only: split-suspends\sbj [symmetric] suspends\sbj)
  apply (simp add: split-suspends\sbj)
  done

from ts\sb neq-i-j j-bound
have ts-A-j: ts-A-j = (hd-prog p\sb (ys @ Write\sb True a’ sop v A’ L’ R’ W’# zs), is\sb, \varnothing\sbj | (dom \varnothing\sbj – read-tmps (ys @ Write\sb True a’ sop v A’ L’ R’ W’# zs)), (), D\sbj, acquired True ?take-sb\sbj O\sbj, release ?take-sb\sbj (dom S\sb) R\sbj)
  by (simp add: split-suspends\sbj)

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\sb sbj = p\sb.
then
have lp: last-prog \( p_j \) \( ?suspends = p_j \)
apply –
apply (rule last-prog-same-append \([\text{where} \ sb=?\text{take-sb}_j]\))
apply (simp only: split-suspends \( j \) \{symmetric\} suspends\( j \))
apply simp
done

from valid-reads \([\text{OF} \ j\text{-bound}'' \ ts_{sb\cdot j}]\)
have reads-consis: reads-consistent False \( O_j \) \( m_{sb} \) \( sb_j \).

from reads-consistent-flush-all-until-volatile-write \([\text{OF} \ (\text{valid-ownership-and-sharing} \ S_{sb} \ (ts_{sb\cdot j}) \ j\text{-bound}'' \ ts_{sb\cdot j}) \ \text{reads-consis}]\)
have reads-consis-m: reads-consistent True \((\text{acquired True} ?\text{take-sb}_j \ O_j)\) \( m \) suspends\( j \)
by (simp add: \( m \) suspends\( j \))

from outstanding-non-write-non-vol-reads-drop-disj \([\text{OF} \ i\text{-bound}'' \ j\text{-bound}'' \ neq-i-j \ ts_{sb\cdot i} \ ts_{sb\cdot j}]\)
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 \([\text{OF} \ this \ \text{reads-consis-m}]\)
have reads-consis-flush-m: reads-consistent True \((\text{acquired True} ?\text{take-sb}_j \ O_j)\)
\((\text{flush} ?\text{drop-sb} \ m)\) suspends\( j \).

from a-unowned-others \([\text{rule-format}, \ \text{OF} \ - \ neq-i-j \ j\text{-bound} \ ts_{sb\cdot j}]\)
obtain a-notin-owns-j: \( a \notin \) acquired True \( ?\text{take-sb}_j \ O_j \) and \( a\)-unshared: \( a \notin \) all-shared \( ?\text{take-sb}_j \)
by auto
from a-not-acquired-others \([\text{rule-format}, \ \text{OF} \ - \ neq-i-j \ j\text{-bound} \ ts_{sb\cdot j}]\)
have a-not-acquired-j: \( a \notin \) all-acquired \( sb_j \)
by auto

from outstanding-non-volatile-refs-owned-or-read-only \([\text{OF} \ j\text{-bound}'' \ ts_{sb\cdot j}]\)
have nvo-j: non-volatile-owned-or-read-only False \( S_{sb} \ O_j \ sb_j \).

have a-no-non-vol-read: \( a \notin \) outstanding-refs is-non-volatile-Read\( sb \) ?drop-sb\( j \)
proof
assume a-in-nvr:a \( \in \) outstanding-refs is-non-volatile-Read\( sb \) ?drop-sb\( j \)

from reads-consistent-drop \([\text{OF} \ \text{reads-consis}]\)
have rc: reads-consistent True \((\text{acquired True} ?\text{take-sb}_j \ O_j)\) \((\text{flush} ?\text{take-sb}_j \ m_{sb})\)
?drop-sb\( j \).

from non-volatile-owned-or-read-only-drop \([\text{OF} \ nvo-j]\)
have nvo-j-drop: non-volatile-owned-or-read-only True \((\text{share} ?\text{take-sb}_j \ S_{sb})\)
\((\text{acquired True} ?\text{take-sb}_j \ O_j)\)
?drop-sb\( j \)

499
by simp

from outstanding-refs-non-volatile-Readsb-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-sb
by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in
simp add: acquired-takeWhile-non-volatile-Write sb)

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-sb
by auto

with read-only-reads-unowned [OF j-bound′′ i-bound neq-i-j [symmetric] ts sb -j ts sb -i]
have a-unowned-sb: a ∉ S 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 sb
[of sbj Oj]
have a ∈ read-only (share ?take-sbj S sb)
by (auto simp add: )
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-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_ab-j]

have h-consis:
  history-consistent φ_jb (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_ab-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

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 ∩ 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)

501
from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]
  last-prog-hd-prog
have hist-consis′: history-consistent \( \varnothing_{\text{sbj}} \) (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\( \text{sb} \) \( \text{ys} = \{} \)
  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\( j \)]
  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
  [simplified split-suspends\( j \)]
  no-volatile-Read\( \text{sb} \)-volatile-reads-consistent [OF no-vol-read], where
\( S = \text{?share-A} \)
obtain is\( j′ \): instrs (Write\( \text{sb} \) True a′ sop v A′ L′ R′ W′# zs) @ is\( \text{sbj} \) =
  is\( j′ \) @ prog-instrs (Write\( \text{sb} \) True a′ sop v A′ L′ R′ W′# zs) \text{ and}
  steps-ys: (?ts-A, ?m-A, ?share-A) \Rightarrow_d^* \text{ y,}
  \( \varnothing_{\text{sbj}} \mid \text{ (dom } \varnothing_{\text{sbj}} \text{ - read-tmps (Write}\( \text{sb} \) True a′ sop v A′ L′ R′ W′# zs)) \text{ ),(},
  \( D_j \lor \text{ outstanding-refs is-volatile-Write}\( \text{sb} \) \text{ ys } \neq \{} \), \text{ acquired True y s}
  (acquired True ?take-sb\( \text{j} \) \( \mathcal{O}_j \),\( \mathcal{R}_j \)),]
, flush ys ?m-A,
  share ys ?share-A)
(is (\( \cdot,\cdot,\cdot \) \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\( \text{sbj} \) ((ys @ [Write\( \text{sb} \) True a′ sop v A′ L′ R′ W′]) @ zs)
  by simp
from causal-program-history-suffix [OF this]
have cph′: causal-program-history is\( \text{sbj} \) zs.
interpret causal\( j \): causal-program-history is\( \text{sbj} \) zs by (rule cph′)
from causal\( j \), causal-program-history [of [], simplified, OF refl] is\( j′ \)
obtain is\( j″ \)
  where is\( j″ \) = Write True a′ sop v A′ L′ R′ W′#is\( j″ \) \text{ and}
is\( j″ \): instrs zs @ is\( \text{sbj} \) = 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 = (last-prog (hd-prog p_j (Write sb True a’ sop v A’ L’ R’ W’# zs)))
y, is'_j',
  \( \vartheta_{sbj} \vdash (\text{dom } \vartheta_{sbj} - \text{read-tmps (Write sb True a’ sop v A’ L’ R’ W’# zs)}))
  D_j \lor \text{outstanding-refs is-volatile-Write sb ys} \neq \{\},
  \text{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:
\( \forall i < \text{length } ?\text{ts-ys}. j \neq i \rightarrow (\text{let } (O_i) = \text{map owned } ?\text{ts-ys}!i \text{ in } a' \notin O_i) \)
  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-unquired-by-others:
\( \forall j < \text{length } (\text{map } O-sb \ ts_{sb}). i \neq j \rightarrow 
  (\text{let } (O_j, sb_j) = \text{map } O-sb \ ts_{sb}! j
    \text{ in } A \cap \text{all-acquired sb}_j = \{\})\)
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 \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
      from j-bound' have j-bound'''': j < length ts
        by simp
    qed
    qed
  }
from conflict obtain \( a' \) where
\[ a' \text{-in}: a' \in A \quad \text{and} \]
\[ a' \text{-in-j}: a' \in \text{all-acquired } s_b j \]
by auto

let \(?\text{take-sb}_j = (\text{takeWhile } (\text{Not} \circ \text{is-volatile-Write}_{s_b}) s_b j)\) 
let \(?\text{drop-sb}_j = (\text{dropWhile } (\text{Not} \circ \text{is-volatile-Write}_{s_b}) s_b j)\) 

{\text{from ts-sim [rule-format, OF } j\text{-bound'] ts}_{s_b-j} j\text{-bound''}}

obtain \( p_j \text{suspends}_i s_{sbj} D_{sbj} D_j \vartheta_{sbj} i s_j \text{ where} \)
\( ts_{sb-j}: ts_{sb} \upharpoonright j = (p_j, i s_{sbj}, \vartheta_{sbj}, s_b j, D_{sbj}, O_j, R_j) \text{ and} \)
suspends$_j$: suspend$_j = ?\text{drop-sb}_j \text{ and} \)
is$_j$: instrs suspend$_s @ i s_{sbj} = i s_j @ \text{progs-instrs suspend$_s$ and} \)
\( D_j: D_{sbj} = (D_j \lor \text{outstanding-refs is-volatile-Write}_{s_b} s_b j \neq \{\}) \text{ and} \)
\( ts_j: ts!j = (\text{hd-prog } p_j \text{suspends}_s, i s_j, \vartheta_{sbj} \upharpoonright (\text{dom } \vartheta_{sbj} - \text{read-tmps suspend$_s$}), (), \)
\( D_j, \text{acquired } \text{True } ?\text{take-sb}_j O_j, \text{release } ?\text{take-sb}_j (\text{dom } S_{sb}) R_j) \)
apply (cases ts$_{sb}!j$)
apply (force simp add: Let-def)
done

{\text{from } a'\text{-in-j all-acquired-append [of } ?\text{take-sb}_j ?\text{drop-sb}_j\]}
have \( a' \in \text{all-acquired } ?\text{take-sb}_j \lor a' \in \text{all-acquired } \text{suspend}_s \)
by (auto simp add: suspend$_s$)
thus False
proof
assume \( a' \in \text{all-acquired } ?\text{take-sb}_j \)
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: \( a' \in \text{all-acquired } \text{suspend}_s \)
from split-all-acquired-in [OF conflict-drop]
show False
proof
assume \( \exists{\text{sop}} a'' v \ y s A L R W. \)
suspend$_s$ = \( y s @ \text{Write}_{sb} \text{ True } a'' \text{sop } v A L R W# zs \land a' \in A \)
then
obtain \( a'' \text{sop' } v' y s A' L' R' W' \text{ where} \)
split-suspend$_s$: suspend$_s$ = \( y s @ \text{Write}_{sb} \text{ True } a'' \text{sop' } v' A' L' R' W'# zs \)
(is suspend$_s$ = ?suspend$\_s$) and
\( a'\text{-A'}: a' \in A' \)
by auto

{\text{from direct-memop-step.WriteVolatile [where } \vartheta = \vartheta_{sb} \text{ and } m=\text{flush } ?\text{drop-sb } m\]}
have (Write True a (D, f) A L R W # i s$_{sb}^\prime$, \( \vartheta_{sb}, (), \text{flush } ?\text{drop-sb } m, D_{sb}, \text{acquired } \text{True } sb O_{sb}, \)
release sb (dom S$_{sb}) R_{sb}$, \share ?drop-sb S) →
\[
(i_{sb}', \varnothing_{sb}, ()), (\text{flush} \ ?\text{drop-sb} \ m)(a := f \varnothing_{sb}), \text{True, acquired True} \text{ sb } \mathcal{O}_{sb} \cup A - R,\text{Map.empty},
\]
\[
\text{share } ?\text{drop-sb } \mathcal{S} \oplus \mathcal{W} \ominus R \ominus A \text{ L}).
\]

from direct-computation.concurrent-step.Memop [OF i-bound-ts' [simplified is_{sb}] ts'\i \text{ [simplified is_{sb}] this [simplified is_{sb}]]}

have store-step: (?ts', \text{flush} ?\text{drop-sb} \ m, \text{share} ?\text{drop-sb} \mathcal{S} \ominus \mathcal{W} \ominus R \ominus A \text{ L})

(is - \Rightarrow (\text{ts-A}, ?\text{m-A}, ?\text{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} \text{ sb}_j.

then have cph: causal-program-history is_{sbj} ?\text{suspends}

apply --

apply (rule causal-program-history-suffix [where sb=?\text{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,

\varnothing_{sbj} |^i (\text{dom } \varnothing_{sbj} - \text{read-tmps (ys @ Write_{sb} True a'' sop' v' A' L' R' W'\# zs)}), (), D_j, \text{acquired True } ?\text{take-sb}_j \mathcal{O}_{j}, \text{release } ?\text{take-sb}_j (\text{dom } \mathcal{S}_{sb}) \mathcal{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 \text{ sb_j} = p_j.

then

have lp: last-prog p_j ?\text{suspends} = p_j

apply --

apply (rule last-prog-same-append [where sb=?\text{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 \mathcal{O}_j m_{sb} \text{ sb_j},

505
from reads-consistent-flush-all-until-volatile-write \([\text{OF} \ \text{valid-ownership-and-sharing} \ S_{sb} \ t_{sb}]\)
\(\text{j-bound}''\)
\(t_{sb} \rightarrow \text{reads-consis}\)
\[\text{have reads-consis-m: reads-consistent True (acquired True} \ ?\text{take-sb}_j \ O_j) \text{ m suspends}_j\]
\text{by (simp add: m suspends}_j\)

from outstanding-non-write-non-vol-reads-drop-disj \([\text{OF} \ i\text{-bound} \ \text{j-bound}'' \ \text{neq-i-j} \ t_{sb} \rightarrow \text{ts}_{sb} \rightarrow j]\)
\text{have outstanding-refs is-Write}_{sb} \ ?\text{drop-sb} \ \cap \ \text{outstanding-refs is-non-volatile-Read}_{sb} \text{ suspends}_j = \{\}\n\text{by (simp add: suspends}_j\)
\text{from reads-consistent-flush-independent \([\text{OF} \ this \ \text{reads-consis-m}]\)\n\text{have reads-consis-flush-m: reads-consistent True (acquired True} \ ?\text{take-sb}_j \ O_j) \text{ (flush} \ ?\text{drop-sb} m) \text{ suspends}_j.\]

from a-unowned-others [rule-format, \text{OF} \ \text{neq-i-j} \ j\text{-bound} \ \text{ts}_{sb} \rightarrow j] \text{obtain a-notin-owns-j: a} \notin \text{acquired True} \ ?\text{take-sb}_j \ O_j \text{ and a-unshared: a} \notin \text{all-shared} \ ?\text{take-sb}_j\n\text{by auto}\n\text{from a-not-acquired-others [rule-format, \text{OF} \ \text{neq-i-j} \ j\text{-bound} \ \text{ts}_{sb} \rightarrow j] \text{have a-not-acquired-j: a} \notin \text{all-acquired sb}_j\n\text{by auto}\n
from outstanding-non-volatile-refs-owned-or-read-only \([\text{OF} \ \text{j-bound}'' \ \text{ts}_{sb} \rightarrow j]\)
\text{have nvo\text{-j: non-volatile-owned-or-read-only False} \ S_{sb} \ O_j \text{ sb}_j.}\n\text{have a-no-non-vol-read: a} \notin \text{outstanding-refs is-non-volatile-Read}_{sb} \ ?\text{drop-sb j}\n\text{proof}\n\text{assume a-in-nvr:a} \in \text{outstanding-refs is-non-volatile-Read}_{sb} \ ?\text{drop-sb j}\n
from reads-consistent-drop \([\text{OF} \ \text{reads-consis}]\)
\text{have rc: reads-consistent True (acquired True} \ ?\text{take-sb}_j \ O_j) \text{ (flush} \ ?\text{take-sb}_j \ m_{sb}) \ ?\text{drop-sb}.\n
from non-volatile-owned-or-read-only-drop \([\text{OF} \ \text{nvo-j}]\)
\text{have nvo\text{-j-drop: non-volatile-owned-or-read-only True} \ (\text{share} \ ?\text{take-sb}_j \ S_{sb}) \ ?\text{drop-sb}_j}\n\text{by simp}\n
from outstanding-refs-non-volatile-Read\text{sb-all-acquired} \([\text{OF} \ \text{rc this a-in-nvr}]\)
\text{have a-owns-acq-ror: a} \in \ O_j \ \cup \ \text{all-acquired sb}_j \ \cup \ \text{read-only-reads} \ (\text{acquired True} \ ?\text{take-sb}_j \ O_j) \ ?\text{drop-sb}_j\n\text{by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in simp add: acquired-takeWhile-non-volatile-Write}_{sb})\n\text{have a-unowned-j: a} \notin \ O_j \ \cup \ \text{all-acquired sb}_j\n\text{proof (cases a} \in \ O_j)\n\text{case False with a-not-acquired-j show} \ \text{thesis by auto}\n
506
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-sb? S\sb j) ?drop-sb? S\sb j
  by auto

with read-only-reads-unowned [OF j-bound'' i-bound neq-i-j [symmetric] Ts\sb j Ts\sb i]
have a-unowned-sb: a /∈ O_{\sb j} ∪ all-acquired sb
  by auto

have consis-j-drop: sharing-consistent (share ?take-sb? S\sb j) (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
  [of sb\sb j O\j]
have a ∈ read-only (share ?take-sb? S\sb j)
  by (auto)
from read-only-share-all-shared [OF this] a-unshared
have a ∈ read-only S\sb j
  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 j)
  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)

      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 $p_j$ sbj$_j$) ?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$_j$ \ intersection 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 $\theta_{sbj}$ (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]
    have no-vol-read: outstanding-refs is-volatile-Read$_{sb}$ ys = {} 
  by (auto simp add: outstanding-refs-append suspends$_j$ [symmetric] split-suspends$_j$ )

508

obtain \( \text{is}_j' R_j' \) where \( \text{is}_j' \): instrs (Write\_sb\_True a'' sop' v' A' L' R' W' # zs) @ \( \text{is}_\text{sbj} = \)

\( \text{is}_j' @ \text{prog-instrs} (Write\_sb\_True a'' sop' v' A' L' R' W' # zs) \) and steps-ys: (?ts-A, ?m-A, ?share-A) \( \Rightarrow \_d^* \)

(\( ?\text{ts-A}[i] = \) (last-prog (hd-prog p_j (Write\_sb\_True a'' sop' v' A' L' R' W' # zs)) ys,

\( \text{is}_j, \)

\( \_d_{\text{sbj}} \mid (\text{dom} \_d_{\text{sbj}} - \text{read-tmps} (Write\_sb\_True a'' sop' v' A' L' R' W' # zs)) \).() ),

\( D_j \lor \) 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 (\(_\cdot\cdot\cdot\) \( \Rightarrow \_d^* \) (?ts-ys, ?m-ys, ?share-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 \( \text{is}_\text{sbj} \) ((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 \( \text{is}_\text{sbj} \) zs.

interpret causal_j: causal-program-history \( \text{is}_\text{sbj} \) zs by (rule cph')

from causal_j, causal-program-history [of [], simplified, OF refl] \( \text{is}_j' \)

obtain \( \text{is}_j'' \)

where \( \text{is}_j' \): \( \text{is}_j' = \text{Write True a'' sop' A' L' R' W' # is}_j'' \) and \( \text{is}_j'' \): instrs zs @ \( \text{is}_\text{sbj} = \text{is}_j'' @ \text{prog-instrs} \) zs

by clarsimp

from j-bound\'"

have j-bound-ys: \( j < \text{length} ?\text{ts-ys} \)

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, \( \text{is}_j' \)

\( \_d_{\text{sbj}} \mid (\text{dom} \_d_{\text{sbj}} - \text{read-tmps} (Write\_sb\_True a'' sop' v' A' L' R' W' # zs)) \).(),\( D_j \lor \) outstanding-refs is-volatile-Write\_sb ys \( \neq \{\}, \) acquired True ys (acquired True ?take-sb\_j O_j, R_j )

by auto

509
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:
\[ \forall i < \text{length } ?ts-ys. j \neq i \rightarrow (\text{let } (\mathcal{O}_i) = \text{map owned } ?ts-ys!i \text{ in } A' \cap \mathcal{O}_i = \{\}) \]
apply cases
apply (fastforce simp add: Let-def is)
done
from a'-in a'-A' A'-unowned [rule-format, of i] neq-i-j i-bound
show False
next
assume \( \exists A L R W y s z s. \)
suspends_{j} = y s @ Ghost_{sb} A L R W # zs \land a' \in A
then
obtain y s z \( A' L' R' W' \) where
split-suspends_{j} = y s @ 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 \( \theta=\theta_{sb} \) and \( m=\text{flush } ?\text{drop-sb m} \)]
have (Write True a (D, f) A L R W# is_{sb},
\( \theta_{sb}, () \), flush ?drop-sb m,D_{sb},acquired True sb \( \mathcal{O}_{sb} \),
release sb (dom \( S_{sb} \)) \( \mathcal{R}_{sb} \),
share ?drop-sb \( S \cap R \) sb,
\( \text{store-step: } (?ts' \in A' \cap \mathcal{O}_i \cap \mathcal{O}_j = \{\}) \)
by (simp add: is_{sb})
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' \in A' \cap \mathcal{O}_i \cap \mathcal{O}_j = \{\})
\( (\text{is} \rightarrow \text{d} (?ts\in A' \cap \mathcal{O}_i \cap \mathcal{O}_j = \{\}) \)
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=\text{take-sb}] )
apply (simp only: split-suspends_{j} [symmetric] suspends_{j})
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), is\_j,
\(\emptyset\)sbj | (dom \(\emptyset\)sbj - read-tmps (ys @ Ghost\_sb A' L' R' W'# zs)), ()D\_j,
acquired True ?take-sb\_j \(\mathcal{O}\_j\),release ?take-sb\_j (dom \(\mathcal{S}\_sb\) \(\mathcal{R}\_j\))
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'' 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 [symmetric] suspends)
apply simp
done

from valid-reads [OF j-bound'' ts\_sb-j]
have reads-consis: reads-consistent False \(\mathcal{O}\_j\) m\_sb sb\_j.

from reads-consistent-flush-all-until-volatile-write [OF (valid-ownership-and-sharing \(\mathcal{S}\_sb\) ts\_sb\-i) j-bound'' ts\_sb-j reads-consis]
have reads-consis-m: reads-consistent True (acquired True ?take-sb\_j \(\mathcal{O}\_j\)) m suspends\_j
by (simp add: m suspends)

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 \(\mathcal{C}\_sb\) \(\mathcal{S}\_sb\) suspends\_j = {}
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-sb\_j \(\mathcal{O}\_j\)) (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 acquired True ?take-sb\_j \(\mathcal{O}\_j\) and a-unshared: a \notin 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 \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-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: reads-consistent True (acquired True ?take-sb_j $O_j$) (flush ?take-sb_j $m_{sb}$) ?\text{drop-sb}_j.

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 $S_{sb}$)
  (acquired True ?take-sb_j $O_j$)
  ?\text{drop-sb}_j
  by 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 sb}_j \cup \text{read-only-reads} (acquired True ?take-sb_j $O_j$) ?\text{drop-sb}_j
  by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in
  simp add: acquired-takeWhile-non-volatile-Write_{sb})

have a-unowned-j: a \notin O_j \cup \text{all-acquired sb}_j
  proof (cases a \in O_j)
  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 \in \text{read-only-reads} (acquired True ?take-sb_j $O_j$) ?\text{drop-sb}_j
  by 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 sb}
  by auto

from sharing-consis [OF j-bound" ts_{sb}-j] sharing-consistent-append [of $S_{sb} O_j$ ?take-sb_j ?\text{drop-sb}_j]
have consis-j-drop: sharing-consistent (share ?take-sb_j $S_{sb}$) (acquired True ?take-sb_j $O_j$) ?\text{drop-sb}_j
  by auto

from \text{read-only-reads-read-only} [OF nvo-j-drop consis-j-drop] a-ror a-unowned-j
  all-acquired-append [of ?take-sb_j ?\text{drop-sb}_j] acquired-takeWhile-non-volatile-Write_{sb}
  [of sb_j $O_j$]
have a \in \text{read-only} (share ?take-sb_j $S_{sb}$)
by (auto)
from read-only-share-all-shared [OF this] a-unshared
have \( a \in \text{read-only} \ S_{sb} \)
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 \( a \in \text{read-only} \ (\text{share}\ sb \ S_{sb}) \)
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 \(?\text{take-sb}_j \ O_j\) \(?m-A\ \text{suspends}_j\)
by (auto simp add: suspends_j)

hence reads-consis-m-A-ys: reads-consistent True (acquired True \(?\text{take-sb}_j \ O_j\) \(?m-A\ y\)
ys
by (simp add: split-suspends_j reads-consistent-append)

from valid-history [OF j-bound" ts_{sb}\-j]
have h-consis:
history-consistent \( \vartheta_{sbj} \ (\text{hd-prog} \ p_j \ (?\text{take-sb}_j @\text{suspends}_j)) \ (?\text{take-sb}_j @\text{suspends}_j) \)
apply (simp only: split-suspends_j [symmetric] suspends_j)
apply simp
done

have last-prog-hd-prog: last-prog (hd-prog p_j \( \text{sb}_j\)) \(?\text{take-sb}_j = \text{(hd-prog p}_j \text{suspends}_j)\)
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 \text{suspends}_j) \(?\text{take-sb}_j = \text{hd-prog p}_j \text{suspends}_j\)
by (simp only: split-suspends_j [symmetric] suspends_j)
moreover
have last-prog (hd-prog p_j \(?\text{take-sb}_j @\text{suspends}_j)) \(?\text{take-sb}_j = \text{last-prog (hd-prog p}_j \text{suspends}_j) \)?\text{take-sb}_j
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 valid-write-sops [OF j-bound" ts_{sb}\-j]
have \( \forall \text{sop}\in\text{write-sops} \ (?\text{take-sb}_j @\text{?suspends}). \text{valid-sop sop} \)
by (simp add: split-suspends \( j \) [symmetric] suspends \( j \))

then obtain valid-sops-take: \( \forall \text{sop} \in \text{write-sops } ?\text{take-sb}_j \). valid-sop sop and
valid-sops-drop: \( \forall \text{sop} \in \text{write-sops } \text{ys} \). valid-sop sop

apply (simp only: write-sops-append )
apply auto
done

from read-tmps-distinct [OF j-bound'' ts_{\text{sb-j}}]
have distinct-read-tmps (?\text{take-sb}_j \cap \text{read-tmps } \text{suspends } \text{ys} ) = \{\}

by (simp add: split-suspends \( j \) [symmetric] suspends \( j \))

then obtain
read-tmps-take-drop: \( \text{read-tmps } ?\text{take-sb}_j \cap \text{read-tmps } \text{suspends } \text{ys} = \{\} \) and

distinct-read-tmps-drop: distinct-read-tmps-drop

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 \( \theta_{\text{sbj}} \) (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 \( \text{sb } \text{ys} = \{\} \)

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 \( j \) ]

\( \text{ts-A-j} \) [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 [simplified split-suspends \( j \) ]

no-volatile-\text{Read}_{\text{sb-ys}}\text{-volatile-reads-consistent [OF no-vol-read], where}

\( \text{S=?'share-A} \)

obtain \( \text{is}_j' \text{ R}_j' \text{ where}

\( \text{is}_j' \): instrs (Ghost_{\text{sb}} A' L' R' W' # zs) @ is_{\text{sbj} j} =

\( \text{is}_j' @ \text{prog-instrs} (\text{Ghost}_{\text{sb}} A' L' R' W' # zs) \) and

\( (\text{?ts-A}_{j:j} = (\text{last-prog } (\text{hd-prog } p_j (\text{Ghost}_{\text{sb}} A' L' R' W' # zs))) ) \) ys,

\( \text{is}_j' \),

\( \theta_{\text{sbj}} \mid (\text{dom } \theta_{\text{sbj}} - \text{read-tmps } (\text{Ghost}_{\text{sb}} A' L' R' W' # zs))),() \),

\( \text{D}_j \lor \text{outstanding-ref } \text{is-volatile-Write}_{\text{sb}} \text{ys} \neq \{\}, \text{acquired } \text{True } \text{ys } (\text{acquired True } \text{?take-sb}_{\text{ys}} \text{O}_{j} \text{,} \text{R}_{j}') \),

\( \text{flush } \text{ys } ?\text{m-ys}, \text{share } \text{ys } ?\text{share-ys} \)

(\text{is } (\cdot,\cdot,\cdot) \Rightarrow d^* (\text{?ts-ys} ,?\text{m-ys},?\text{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 (\(ys \mathbin{@} \{\text{Ghost}_{\text{sb}} A' L' R' W'\}\mathbin{@} zs) by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history is\sbj zs by (rule cph')
from causal,causal-program-history [of [], simplified, OF refl] is\j
obtain is\j''
where is\j': is\j = Ghost A' L' R' W' #is\j'' and
is\j'': instrs zs @ is\sbj = is\j'' @ prog-instrs zs
by clarsimp
from j-bound''
have j-bound-ys: j < length ?ts-ys
by auto
from j-bound-ys neq-i-j
have ts-ys-j: ?ts-ys{j} = (last-prog (hd-prog p\j (Ghost\sb\j A' L' R' W' # zs)) ys, is\j', 
\v\sbj | (dom \v\sbj \mathbin{-} \text{read-tmps (Write}_{\text{sb}} \text{True a'' sop}' v' A' L' R' W' # zs))).(\),D\j 
\vee \text{outstanding-refs is-volatile-Write}_{\text{sb}} ys \neq \emptyset
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 \longrightarrow (\text{let } (O_i) = \text{map owned } ?ts-ys)i \text{ in } A' \cap O_i = \emptyset
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 ?thesis
by (auto simp add: Let-def)
qed
have A-no-read-only-reads-by-others:
\forall j<\text{length } (\text{map } O_{-}\text{sb } t\sb\j). i \neq j \longrightarrow
\text{(let } (O_j, sb_j) = \text{map } O_{-}\text{sb } t\sb\j! j}
in \( A \cap \text{read-only-reads} (\text{acquired True} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}) \ \text{sb}_j)) \)

\( \mathcal{O}_j \)

\((\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}) \ \text{sb}_j) = \{\}\)

\text{proof} –

\[
\begin{align*}
\{ & \text{fix } j \ \mathcal{O}_j \ \text{sb}_j \\
& \text{assume } j \text{-bound: } j < \text{length} \ (\text{map} \ \mathcal{O} \ \text{sb} \ \text{ts}_\text{sb}) \\
& \text{assume } \text{neq-i-j: } i \neq j \\
& \text{assume } \text{ts}_\text{sb}-j: \ (\text{map} \ \mathcal{O} \ \text{sb} \ \text{ts}_\text{sb}) \ l_j = (\mathcal{O}_j, \text{sb}_j) \\
\text{let } & ?\text{take-sb}_j = (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}) \ \text{sb}_j) \\
\text{let } & ?\text{drop-sb}_j = (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}) \ \text{sb}_j) \\
\text{assume } & \text{conflict: } A \cap \text{read-only-reads} (\text{acquired True } ?\text{take-sb}_j \ \mathcal{O}_j) \ ?\text{drop-sb}_j \neq \{\} \\
\text{have } & \text{False} \text{ } \text{proof} – \\
\text{from } & \text{j-bound leq} \\
\text{have } & \text{j-bound”: } j < \text{length} \ \text{ts}_\text{sb} \\
& \text{by auto} \\
\text{from } & \text{j-bound” have } j \text{-bound””}: j < \text{length} \ \text{ts} \\
& \text{by simp} \\
\text{from } & \text{conflict obtain } a' \text{ where} \\
& a' \text{-in: } a' \in A \text{ and} \\
& a' \text{-in-j: } a' \in \text{read-only-reads} (\text{acquired True } ?\text{take-sb}_j \ \mathcal{O}_j) \ ?\text{drop-sb}_j \\
& \text{by auto} \\
\text{from } & \text{ts-sim [rule-format, OF } j \text{-bound””] ts}_\text{sb}-j j \text{-bound””} \\
\text{obtain } & p_j \text{ suspends}_j is_{sbj} D_{sbj} D_j R_j \ \theta_{sbj} is_j \text{ where} \\
& ts_{sb}-j: ts_{sb} ! j = (p_j, is_{sbj}, \ \theta_{sbj}, \ sb_j, D_{sbj}, \mathcal{O}_j, R_j) \ \text{and} \\
& \text{suspends}_j: \text{suspends}_j = ?\text{drop-sb}_j \ \text{and} \\
& is_j: \text{instrs suspends}_j @ is_{sbj} = is_j @ \text{prog-instrs suspends}_j \ \text{and} \\
& D_j: D_{sbj} = (D_j \lor \text{outstanding-refs} \text{ is-volatile-Write} \ \text{sb}_j \neq \{\}) \ \text{and} \\
& ts_j: ts_j = (\text{hd-prog} p_j \text{ suspends}_j, is_j, \\
& \ \theta_{sbj} | (\text{dom} \ \theta_{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) \\
& \text{apply (cases ts}_{sb}-l_j) \\
& \text{apply (force simp add: Let-def)} \\
& \text{done} \\
\text{from } & \text{split-in-read-only-reads [OF } a' \text{-in-j [simplified suspends}_j \text{ [symmetric]]] \text{]} \\
\text{obtain } & t v \ text{ys } zs \text{ where} \\
& \text{split-suspends}_j: \text{suspends}_j = \text{ys @ Read}_{sb} \ \text{False } a' t v \# zs (\text{is } \text{suspends}_j = ?\text{suspends}) \ \text{and} \\
& a' \text{-unacq: } a' \notin \text{acquired True } \text{ys} (\text{acquired True } ?\text{take-sb}_j \ \mathcal{O}_j) \\
& \text{by blast} \end{align*}
\]

516
from direct-memop-step.WriteVolatile [where  θ=θ sb and m=flush ?drop-sb m]
have (Write True a (D, f) A L R W# is sb'),
  θ 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', θ sb', () , (flush ?drop-sb m)(a := f θ sb), True, acquired True sb O sb ∪ A – R.Map.empty,
  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' [i := (p sb, is sb', θ sb, () , True, acquired True sb O sb ∪ A – R.Map.empty)],
  (flush ?drop-sb m)(a := f θ sb), share ?drop-sb S ⊕ W R ⊖ 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 sb j 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-A-j: ?ts-A' j = (hd-prog p j (ys @ Read sb False a' t v# zs), is j,
  θ sb j | (dom θ sb j – 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 $O_j$ $m_{sb_j}$.

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing $S_{sb_j}$ $ts_{sb_j}$ i-bound"

$ts_{sb_j}$ j-bound"

have reads-consis-flush-m: reads-consistent True (acquired True ?take-sb $O_j$) $m$ suspends$_j$

by (simp add: $m$ suspends$_j$)

from outstanding-non-write-non-volatile-reads-drop-disj [OF i-bound j-bound" neq-i-j $ts_{sb_j}$ j-bound"

have outstanding-refs is-Write$_{sb_j}$ ?drop-sb $\cap$ outstanding-refs is-non-volatile-Read$_{sb_j}$

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 $O_j$)

(flush ?drop-sb $m$) suspends$_j$.

from a-unowned-others [rule-format, OF j-bound" neq-i-j $j$-bound $ts_{sb_j}$]

obtain a-notin-owns-j: $a$ $\notin$ acquired True ?take-sb $O_j$ 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 $S_{sb_j}$ $O_j$ sb$_j$.

have a-no-non-vol-read: $a$ $\notin$ outstanding-refs is-non-volatile-Read$_{sb_j}$ ?drop-sb$_j$

proof

assume a-in-nvr:$a$ $\in$ outstanding-refs is-non-volatile-Read$_{sb_j}$ ?drop-sb$_j$

from reads-consistent-drop [OF reads-consis]

have rc: reads-consistent True (acquired True ?take-sb $O_j$) (flush ?take-sb $m_{sb_j}$)

?drop-sb$_j$.

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 $S_{sb_j}$)

(acquired True ?take-sb $O_j$)

?drop-sb$_j$

by simp

from outstanding-refs-non-volatile-Read$_{sb_j}$ all-acquired [OF rc this a-in-nvr]

have a-owns-acq-ror:

$a$ $\in$ $O_j$ $\cup$ all-acquired sb$_j$ $\cup$ read-only-reads (acquired True ?take-sb $O_j$) ?drop-sb$_j$

by (auto dest!: acquired-all-acquired-in all-acquired-takeWhile-dropWhile-in simp add: acquired-takeWhile-nonnons-volatile-Write$_{sb_j}$)

have a-unowned-j: $a$ $\notin$ $O_j$ $\cup$ all-acquired sb$_j$
proof (cases a ∈ O_j)
  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-sb j O_j) ?drop-sb_j
  by auto

with read-only-reads-unowned [OF j-bound” i-bound neq-i-j [symmetric] ts sb-j ts sb-i]
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 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

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 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
  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
valid-sops-drop: ∀sop∈write-sops ys. valid-sop sop
  apply (simp only: write-sops-append )
  apply auto
  done

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
  by (simp only: split-suspends_j [symmetric] suspends_j)

done

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_{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\(_j\) 

\[
\begin{align*}
\text{from} & \quad \text{flush-store-buffer-append} [\\OF j\text{-bound}'''] \quad \text{is}_j [\text{simplified split-suspends}_j] \quad \text{cph} [\text{simplified suspends}_j] \\
ts-A\_j [\text{simplified split-suspends}_j] \quad \text{refl lp} [\text{simplified split-suspends}_j] \quad \text{reads-consis-m-A-ys} \\
hist-consis' [\text{simplified split-suspends}_j] \quad \text{valid-sops-drop} \quad \text{distinct-read-tmps-drop} \\
[\text{simplified split-suspends}_j] \\
\text{no-volatile-Read}_{ab\_volatile-reads-consistent} [\text{OF no-vol-read}], \text{where} \\
S = \text{?share-A} \\
\text{obtain} \quad \text{is}_j' \quad \mathcal{R}_j \quad \text{where} \\
is_j': \text{instrs} (\text{Read}_{ab} \text{False} a' t v\# zs) @ \text{is}_{abj} = \\
is_j' @ \text{prog-instrs} (\text{Read}_{ab} \text{False} a' t v\# zs) \quad \text{and} \\
\text{steps-ys}: (?ts-A, ?m-A, ?share-A) \Rightarrow d^* \\
(?ts-A\_j):= (\text{last-prog} (\text{hd-prog} p_j (\text{Read}_{ab} \text{False} a' t v\# zs))) \text{ys}, \\
is_j', \\
\theta_{abj} |' (\text{dom} \theta_{abj} - \text{read-tmps} (\text{Read}_{ab} \text{False} a' t v\# zs)),(), \\
D_j \land \text{outstanding-refs is-volatile-Write}_{ab} \text{ys} \neq \{\}, \text{acquired True} \text{ys} \\
(\text{acquired True} ?\text{take-sb}_{j} O_j),\mathcal{R}_j') \\
\text{flush} \text{ys} ?m-A, \\
\text{share} \text{ys} ?\text{share-A} \\
(\text{is} (\_,-,-) \Rightarrow d^* (?ts-ys,?m-ys,?\text{shared-ys})) \\
\text{by} \quad \text{(auto)} \\
\text{note} \quad \text{conflict-computation} = \text{rtranclp-trans} [\text{OF rtranclp-r-rtranclp} [\text{OF steps-flush-sb,} \\
\text{OF store-step}\text{]}\text{ steps-ys}]] \\
\text{from} \quad \text{cph} \\
\text{have} \quad \text{causal-program-history} \text{is}_{abj} ((\text{ys} @ [\text{Read}_{ab} \text{False} a' t v]) @ \text{zs}) \\
\text{by} \quad \text{simp} \\
\text{from} \quad \text{causal-program-history-suffix} [\text{OF this}] \\
\text{have} \quad \text{cph}': \text{causal-program-history} \text{is}_{abj} \text{zs}. \\
\text{interpret} \quad \text{causal} j': \text{causal-program-history} \text{is}_{abj} \text{zs} \text{by} \quad \text{(rule cph')} \\
\text{from} \quad \text{causal}\_j.\text{causal-program-history} [\text{of} [] \text{, simplified, OF refl} \text{]} \quad \text{is}_j'' \\
\text{obtain} \quad \text{is}_j'' \\
\text{where} \quad \text{is}_j': \text{is}_j'' = \text{Read} \text{False} a' t\#i\text{s}_j''' \quad \text{and} \\
\text{is}_j'': \text{instrs} \text{zs} @ \text{is}_{abj} = \text{is}_j''' @ \text{prog-instrs} \text{zs} \\
\text{by} \quad \text{clarsimp} \\
\text{from} \quad \text{j-bound'''} \\
\text{have} \quad \text{j-bound-ys}: j \prec \text{length} ?\text{ts-ys} \\
\text{by} \quad \text{auto} \\
\text{from} \quad \text{j-bound-ys neq-i-j} \\
\text{have} \quad \text{ts-ys-j}: (?\text{ts-ys}\_j):= (\text{last-prog} (\text{hd-prog} p_j (\text{Read}_{ab} \text{False} a' t v\# zs))) \text{ys}, \text{is}_j', \\
\theta_{abj} |' (\text{dom} \theta_{abj} - \text{read-tmps} (\text{Read}_{ab} \text{False} a' t v\# zs)),(), \\
D_j \land \text{outstanding-refs is-volatile-Write}_{ab} \text{ys} \neq \{\}, \text{acquired True} \text{ys} (\text{acquired True} ?\text{take-sb}_{j} O_j),\mathcal{R}_j') \\
\text{by} \quad \text{auto} \\
\text{from} \quad \text{safe-reach-safe-rtrancl} [\text{OF safe-reach conflict-computation}] \\
\text{521}
have safe-delayed (?ts-ys,?m-ys,?shared-ys).

from safe-delayedE [OF this j-bound-ys ts-ys-j, simplified isj] 
have a' ∈ acquired True ys (acquired True ?take-sb j) \lor
  a' ∈ 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' ∈ read-only (share ys (share ?drop-sb S ⊕ W R ⊕ A L))
  by auto
from a'\'-in
have a'\'-not-ro: a' \∉ read-only (share ?drop-sb S ⊕ W R ⊕ A L)
  by (auto simp add: in-read-only-convs)

have a' ∈ O \_j ∪ all-acquired sb\_j
proof -
  { assume a-notin: a' \∉ O \_j ∪ all-acquired sb\_j
  from weak-sharing-consist [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) suspends\_j
    by (auto simp add: suspends\_j)
  with split-suspends\_j
  have weak-consis: weak-sharing-consistent (acquired True ?take-sb 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 \_sb \_j [of sb\_j O \_j]
  all-acquired-append [of ?take-sb j ?drop-sb j]
  have a' \∉ acquired True (takeWhile (Not o is-volatile-Write\_sb\_sb) sb\_j) O \_j ∪ all-acquired ys
    by auto
  from read-only-share-unowned [OF weak-consis this a'\'-ro]
  have a' ∈ 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

522
have $A \cap \text{all-acquired } s_b^{j} = \{\}$
  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 $A \cap O_j^{j} = \{\}$
  by (auto simp add: Let-def dest: all-shared-acquired-in)
moreover note $a^{j}$-in
ultimately
show False
  by auto
qed
thus $\?\text{thesis}$
  by (auto simp add: Let-def)
qed

have \text{valid-own}$^{j}$: valid-ownership $S_{sb}^{j}$ $ts_{sb}^{j}$
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only $S_{sb}^{j}$ $ts_{sb}^{j}$'
proof
  from outstanding-non-volatile-refs-owned-or-read-only [OF $i$-bound $ts_{sb}^{i}$-$i$]
  have non-volatile-owned-or-read-only False $S_{sb}^{j}$ $O_j^{j}$ (sb @ [Write$_{sb}^{j}$ True a (D,$f$) ($f$ $\vartheta$$_{sb}^{j}$) 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 $\?\text{thesis}$ by (simp add: $ts_{sb}^{j}$-$sb^{j}$ $O_j^{j}$ $S_{sb}^{j}$')
qed
next
show outstanding-volatile-writes-unowned-by-others $ts_{sb}^{j}$'
proof (unfold-locales)
  fix $i_1$ $p_1$ $is_1^{j}$ $D_1$ $xs_1$ $sb_1$ $p_j$ $is_j^{j}$ $D_j$ $xs_j$ $sb_j$
  assume $i_1$-$bound$: $i_1$ < length $ts_{sb}^{j}$'
  assume $j$-$bound$: $j$ < length $ts_{sb}^{j}$'
  assume $i_1$-$j$: $i_1$ $\neq$ $j$
  assume $ts$-$i_1$: $ts_{sb}^{j}$ $\cap$ $S_{sb}^{j}$ = ($p_1$,is$_1^{j}$,$xs_1$,$sb_1$,$D_1$,$O_j$,$R_j$)
  assume $ts$-$j$: $ts_{sb}^{j}$ $\cap$ $S_{sb}^{j}$ = ($p_j$,is$_j^{j}$,$xs_j$,$sb_j$,$D_j$,$O_j$,$R_j$)
  show $(O_j^{j} \cup \text{all-acquired } s_b^{j}) \cap \text{outstanding-refs is-volatile-Write}_{sb}^{j} s_b^{j} = \{\}$
proof (cases $i_1$-$j$)
case True
  with $i_1$-$j$ have $i$-$j$: $i \neq j$
    by simp
from $j$-$bound$ have $j$-$bound$": $j$ < length $ts_{sb}^{j}$
  by (simp add: $ts_{sb}^{j}$')
hence $j$-$bound$": $j$ < length (map owned $ts_{sb}^{j}$)
  by simp
from $ts$-$j$ $i$-$j$ have $ts$-$j$": $ts_{sb}^{j}$ $\cap$ $S_{sb}^{j}$ = ($p_j$,is$_j^{j}$,$xs_j$,$sb_j$,$D_j$,$O_j$,$R_j$)
  by (simp add: $ts_{sb}^{j}$')
from a-unowned-others [rule-format, OF - $i$-$j$] $i$-$j$ $ts$-$j$ $j$-bound
obtain $a^{j}$-in: a $\notin$ acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{sb}^{j}$ $sb_j$) $O_j$) and
a-unshared: a \notin all-shared (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb_j) \\
by (auto simp add: \text{Let-def} ts_{sb})
from a-not-acquired-others \ [\text{rule-format}, \text{OF} - i-j] i-j \text{ ts-j j-bound} \\
have a-notin-acq: a \notin all-acquired sb_j \\
by (auto simp add: \text{Let-def} ts_{sb})
from outstanding-volatile-writes-unowned-by-others \\
[\text{OF i-bound j-bound'} i-j ts_{sb-i} ts-j'] \\
have (\mathcal{O}_j \cup \text{all-acquired sb}_j) \cap \text{outstanding-refs} \text{is-volatile-Write}_{sb} sb = \{\}.
with ts-i_j a-notin-j a-unshared a-notin-acq True i-bound show ?thesis \\
by (auto simp add: \text{let-def} ts_{sb})
ultimately \\
show ?thesis

next \\
case False \\
note i_1-i = this \\
from i-bound have i-bound': i_1 < \text{length} ts_{sb} \\
by (simp add: ts_{sb})
from ts-i False have ts-i': ts_{sb}[i_1] = (p_{i_1}, i_{i_1}, x_{i_1}, sb_{i_1}, \mathcal{D}_{i_1}, \mathcal{O}_{i_1}, \mathcal{R}_{i_1}) \\
by (simp add: ts_{sb}) \\
show ?thesis \\
proof (cases j=i) \\
case True \\
from i-bound' have i-bound'': i_1 < \text{length} (\text{map} \text{ owned} ts_{sb}) \\
by simp \\
from outstanding-volatile-writes-unowned-by-others \\
[\text{OF i-bound'} i-bound i-i_1 ts-i ts_{sb-i}] \\
have (\mathcal{O}_{sb} \cup \text{all-acquired sb}) \cap \text{outstanding-refs} \text{is-volatile-Write}_{sb} sb_1 = \{\}. \\
moreover \\
from A-unused-by-others \ [\text{rule-format}, \text{OF} - \text{False} \ [\text{symmetric}]] \text{False ts-i_1 i-bound} \\
have A \cap \text{outstanding-refs} \text{is-volatile-Write}_{sb} sb_1 = \{\} \\
by (auto simp add: \text{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 < \text{length} ts_{sb} \\
by (simp add: ts_{sb}' ) \\
from ts-j False have ts-j': ts_{sb}[j] = (p_j, i_s, x_s, sb_j, \mathcal{D}_j, \mathcal{O}_j, \mathcal{R}_j) \\
by (simp add: ts_{sb}')
from outstanding-volatile-writes-unowned-by-others \\
[\text{OF i-bound'} j-bound' i-j ts-i' ts-j'] \\
show (\mathcal{O}_j \cup \text{all-acquired sb}_j) \cap \text{outstanding-refs} \text{is-volatile-Write}_{sb} sb_1 = \{\} .
qed 
qed 
qed
next

show ownership-distinct $t_{sb}'$

proof –

have $\forall j < \text{length } t_{sb}$. $i \neq j \rightarrow$

(let $(p_j, is_j, t_j, sb_j, D_j, O_j, R_j) = t_{sb} ! j$

in $(O_{sb} \cup \text{all-acquired } sb') \cap (O_j \cup \text{all-acquired } sb_j) = \{\}$)

proof –

{
  fix $j$ $p_j$ is $j$ $O_j$ $R_j$ $D_j$ $O_{sb}$ $R_{sb}$
  assume neq-i-j: $i \neq j$
  assume j-bound: $j < \text{length } t_{sb}$
  assume $t_{sb}-j$: $t_{sb} ! j = (p_j, is_j, t_j, sb_j, D_j, O_j, R_j)$
  have $(O_{sb} \cup \text{all-acquired } sb') \cap (O_j \cup \text{all-acquired } sb_j) = \{\}$

proof –

{
  fix $a'$
  assume a'-in-i: $a' \in (O_{sb} \cup \text{all-acquired } sb')$
  assume a'-in-j: $a' \in (O_j \cup \text{all-acquired } sb_j)$
  have False

proof –

  from a'-in-i have $a' \in (O_{sb} \cup \text{all-acquired } sb) \lor a' \in A$
  by (simp add: sb'-all-acquired-append)
  then show False

proof

  assume $a' \in (O_{sb} \cup \text{all-acquired } sb)$
  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': $j < \text{length } (\text{map owned } t_{sb}')$

using j-bound by auto

from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb}-j j-bound
obtain $A \cap \text{acquired } \text{True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) O_j = \{\}$ and
$A \cap \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) = \{\}$

by (auto simp add: Let-def)

moreover

from A-unacquired-by-others [rule-format, OF - neq-i-j] ts_{sb}-j j-bound
have $A \cap \text{all-acquired } sb_j = \{\}$

by auto

ultimately

show ?thesis

using a'-in-j

by (auto dest: all-shared-acquired-in)

qed

qed

}
then show ?thesis by auto

cqd
\begin{verbatim}
} then show ?thesis by (fastforce simp add: Let-def)
qed

from ownership-distinct-nth-update [OF i-bound ts_{sb\cdot 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} \emptyset_n sb_{n} p_m is_{m} O_{m} R_{m} D_{m} \emptyset_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}, \emptyset_n, sb_{n}, D_{n}, O_{n}, R_{n})
and mth: ts_{sb}'!m = (p_m, is_{m}, \emptyset_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 \text{all-acquired sb}_{m}) \cap
\text{read-only-reads \{acquired True (takeWhile (Not o is-volatile-Write_{sb}) sb_{n}) O_{n}\}}
\{\text{dropWhile (Not o 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, 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}]
moreover
from A-no-read-only-reads-by-others [rule-format, OF - neq-n-i [symmetric]] n-bound'
nth'
have A \cap \text{read-only-reads \{acquired True (takeWhile (Not o is-volatile-Write_{sb}) sb_{n}) O_{n}\}}
\{\text{dropWhile (Not o is-volatile-Write_{sb}) sb_{n}} = \}
by auto
ultimately
show ?thesis
using True ts_{sb'-i} nth' mth n-bound' m-bound'
by (auto simp add: ts_{sb}' O_{sb}' 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}, \emptyset_m, sb_{m}, D_{m}, O_{m}, R_{m})
by (auto simp add: ts_{sb}')
show ?thesis
proof (cases n=i)
\end{verbatim}
**case True**

**note** read-only-reads-unowned [OF i-bound m-bound’ neq-m-i [symmetric] ts\_sb\_i mth’]  

**then show** ?thesis  
**using** True neq-m-i ts\_sb\_i nth mth n-bound’ m-bound’  
**apply** (case-tac outstanding-refs (is-volatile-Write\_sb) sb = \{\})  
**apply** (clarsimp simp add: outstanding-vol-write-take-drop-append ts\_sb sb’ O\_sb’)+

**done**

**next**

**case False**  
**with** n-bound nth i-bound **have** nth’: ts\_sb\_i n = (p\_n, i\_n, 0\_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**

**have** valid-hist’: valid-history program-step ts\_sb’  
**proof**  
**from** valid-history [OF i-bound ts\_sb\_i]  
**have** history-consistent 0\_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 0\_sb (hd-prog p\_sb (sb@[Write\_sb True a (D,f) (f 0\_sb) A L R W]))  
(sb@[Write\_sb True a (D,f) (f 0\_sb) A L R W]))  
**apply**  
**apply** (rule history-consistent-appendI)  
**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’ 0\_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 True a (D,f) (f 0\_sb) A L R W]).  
**from** valid-reads-nth-update [OF i-bound this]  
**show** ?thesis **by** (simp add: ts\_sb’ sb’ O\_sb’)

**qed**

**have** valid-sharing’: valid-sharing S\_sb’ ts\_sb’  
**proof** (intro-locales)

527
from outstanding-non-volatile-writes-unshared [OF i-bound ts\_sb\_i]
have non-volatile-writes-unshared $S_{sb}$ (sb $\in$ [Write\_sb True a (D,f) (f $\circ$ 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' sb' $S_{sb}'$
next
from sharing-consis [OF i-bound ts\_sb\_i]
have consis': sharing-consistent $S_{sb}$ $O_{sb}$ sb.
from A-shared-owned
have A $\subseteq$ dom (share ?drop-sb $S$) $\cup$ acquired True sb $O_{sb}$
  by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write\_sb)
moreover have dom (share ?drop-sb $S$) $\subseteq$ dom $S$ $\cup$ dom (share sb $S_{sb}$)
proof
  fix a'
  assume a'-in: a' $\in$ dom (share ?drop-sb $S$)
  from share-unshared-in [OF a'-in]
  show a' $\in$ dom $S$ $\cup$ dom (share sb $S_{sb}$)
  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 $S_{sb}$)
      by auto
    thus ?thesis
      by simp
  next
    assume a' $\in$ dom $S$ $\land$ a' / $\notin$ all-unshared ?drop-sb
    thus ?thesis by auto
  qed
qed
ultimately
have A-subset: A $\subseteq$ dom $S$ $\cup$ dom (share sb $S_{sb}$) $\cup$ acquired True sb $O_{sb}$
  by auto
with A-unowned-by-others
have A $\subseteq$ dom (share sb $S_{sb}$) $\cup$ acquired True sb $O_{sb}$
proof
  {  
    fix x
    assume x-A: x $\in$ A
    have x $\in$ dom (share sb $S_{sb}$) $\cup$ acquired True sb $O_{sb}$
    proof
      {  
        assume x $\in$ 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
528
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 $\theta_{sb}$) 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: $ts_{sb}$ $O_{sb}$)
next
from unowned-shared-nth-update [OF i-bound $ts_{sb}$-i subset-refl]
show unowned-shared $S_{sb}'$ $ts_{sb}'$
  by (simp add: $ts_{sb}'$ $ts_{sb}$ $O_{sb}'$ $S_{sb}'$

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 $\theta_{sb}$) 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: $ts_{sb}'$ $ts_{sb}$ $S_{sb}'$
q  

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 $\theta_{sb}$) 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 $is_{sb}'$ $\cap$ read-tmps (sb @ [Write sb True a (D,f) (f $\theta_{sb}$) A L R W]) ={1}
  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: $ts_{sb}'$ $sb'$
}
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 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} True a (D,f) (f \circ_{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\']

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 \cap load-tmps is_{sb}' = \{\} and
  dd-is: data-dependency-consistent-instrs (dom \circ_{sb}') is_{sb}'
  by (auto simp add: is_{sb} \circ_{sb}')

from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i] D-indep

have load-tmps is_{sb}' \cap \bigcup (fst t write-sops (sb@ [Write_{sb} True a (D,f) (f \circ_{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}' 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 \circ_{sb} = \{\}
  by (auto simp add: is_{sb})

from load-tmps-fresh-nth-update [OF i-bound this]

show \?thesis by (simp add: ts_{sb}' \circ_{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 outstanding-refs is-volatile-Write_{sb} (sb@[Write_{sb} True a (D,f) (f \circ_{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
have valid-program-history\(\vdash\) 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\(\vdash\) causal-program-history \(is_{sb}\)′ (sb@\([Write_{sb}\ True\ a\ (D,f)\ A\ L\ R\ W]\)
  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 \(p_{sb}\) (sb@ \([Write_{sb}\ True\ a\ (D,f)\ A\ L\ R\ W]\) = \(p_{sb}\)
  by (simp add: last-prog-append-Write_{sb})
from valid-program-history-nth-update [OF i-bound causal\(\vdash\)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 \(\circ\) is-volatile-Write_{sb}) sb = sb
  by (auto simp add: outstanding-refs-conv)
from True have suspend-nothing: dropWhile (Not \(\circ\) 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 = Write True a (D,f) A L R W# \(is_{sb}\)′
  by (simp add: is_{sb})
with suspends-empty ts-i
have ts-i: \(ts!i\) = (\(p_{sb}\), Write True a (D,f) A L R W# \(is_{sb}\)′, \(Θ_{sb}\).(,\(D\), acquired True ?take-sb \(O_{sb}\), release ?take-sb (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
[OF True i-bound \(ts_{sb}\)-i]

from True
have drop-app: dropWhile (Not \(\circ\) is-volatile-Write_{sb})
  (sb@\([Write_{sb}\ True\ a\ (D,f)\ A\ L\ R\ W]\) =
  \([Write_{sb}\ True\ a\ (D,f)\ A\ L\ R\ W]\)
  by (auto simp add: outstanding-refs-conv)
have \((ts_{sb}', m_{sb}, S_{sb}') \sim (ts, m, S)\)

apply (rule sim-config.intros)

apply \((\text{simpl add: } m \text{ flush-commute } ts_{sb}' \theta_{sb}' O_{sb}' R_{sb}' sb')\)

using share-all-until-volatile-write-Write-commute

[OF i-bound ts_{sb}-i [simplified is_{sb}]]

apply \((\text{clarsimp simp add: } S_{sb}' ts_{sb}' sb' O_{sb}' R_{sb}' \theta_{sb}')\)

using leq

apply \((\text{simpl add: } ts_{sb}')\)

using i-bound i-bound'

apply \(\text{clarsimp simp add: Let-def nth-list-update drop-app ts}_{sb}' sb' O_{sb}' R_{sb}' \theta_{sb}' D_{sb}' \text{ outstanding-refs-append takeWhile-tail flush-all split: if-split-asm }\)

done

ultimately show \(?thesis\)

using valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct'

valid-sops'

valid-dd' load-tmps-fresh' enough-flushes'

valid-program-history' valid' m_{sb}' S_{sb}'

by auto

next

case False

then obtain \(r\) where \(r\in\text{ set } sb\) and \(\text{volatile-r: is-volatile-Write}_{sb}\ r\)

by (auto simpl add: outstanding-refs-conv)

from takeWhile-dropWhile-real-prefix

[OF \(r\)-in, of \((\text{Not } \circ \text{is-volatile-Write}_{sb})\), simplified, OF \(\text{volatile-r}\)]

obtain \(a' v' sb'' A'' L'' R'' W'' sop' where\)

sb-split: \(sb = \text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) sb @ \text{Write}_{sb} True a' sop' v' A'' L'' R'' W'' # sb''\)

and

drop: dropWhile \((\text{Not } \circ \text{is-volatile-Write}_{sb}) sb = \text{Write}_{sb} True a' sop' v' A'' L'' R'' W'' # sb''\)

apply (auto)

subgoal for \(y\) \(ys\)

apply (case-tac \(y\))

apply auto

done

done

from drop suspends have suspends: suspends = \(\text{Write}_{sb} True a' sop' v' A'' L'' R'' W'' # sb''\)

by simp

have \((ts, m, S) \Rightarrow_d^* (ts, m, S)\) by auto

moreover

note flush-commute =

flush-all-until-volatile-write-append-unflushed [OF False i-bound ts_{sb}-i]
have \( \text{Write}_{sb} \) True \( a' \) sop' \( \nu' \) 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: S\( _{sb} \) ' 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

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

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}′)
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 θ_{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}′ θ_{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 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}′
    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}′)

534
from unowned-shared-nth-update [OF i-bound ts sb-i subset-refl]
show unowned-shared $S_{sb} \cap ts_{sb}'$ by (simp add: $ts_{sb}' \cap \emptyset \cap S_{sb}'$

next
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound, of []]
show no-outstanding-write-to-read-only-memory $S_{sb} \cap ts_{sb}'$
by (simp add: $S_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$

qed

have tmpls-distinct $'$: tmpls-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: $is_{sb}' \cap ts_{sb}' \cap \emptyset \cap \emptyset \cap S_{sb}'$
next
from read-tmps-distinct [OF i-bound ts sb-i]
have distinct-read-tmps [] by (simp add: $ts_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$
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 $is_{sb}' \cap \emptyset \cap \emptyset \cap \emptyset \cap S_{sb}'$
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: $ts_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$
next
from load-tmps-write-tmps-distinct [OF i-bound ts sb-i]
load-tmps-write-tmps-distinct [OF i-bound ts sb-i]
have load-tmps $is_{sb}' \cap \emptyset \cap \emptyset \cap \emptyset \cap S_{sb}'$
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: $ts_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$

qed

have valid-sops $'$: valid-sops $ts_{sb}'$
proof —
from valid-store-sops [OF i-bound ts sb-i]
obtain
valid-store-sops $'$: $\forall sop \in store-sops \ is_{sb}' \cdot valid-sop sop$
by (auto simp add: $is_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$
from valid-sops-nth-update [OF i-bound - valid-store-sops', where $sb= []$
show $\$thesis by (auto simp add: $ts_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$

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 $\emptyset _{sb}'$) $is_{sb}'$
by (auto simp add: $is_{sb}' \cap ts_{sb}' \cap \emptyset \cap S_{sb}'$
from load-tmps-write-tmps-distinct [OF i-bound ts sb-i]
have load-tmps $is_{sb}' \cap \emptyset \cap \emptyset \cap \emptyset \cap S_{sb}'$
by (clarsimp)

535
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 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 ts sb' sb' sb θ sb')
qed

from enough-flushs-nth-update [OF i-bound, where sb=[]]
have enough-flushs': enough-flushs ts sb'
proof -
from enough-flushs [OF i-bound ts sb]
have load-tmps is 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' sb)
qed

from is-sim have is: is = Fence # is sb'
by (simp add: suspends sb is sb)
  with ts-i
  have ts-i: ts li = (p sb, Fence # is sb', θ sb, (), D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb)
    by (simp add: suspends sb)
from direct-memop-step.Fence
have (Fence # is sb',
  θ sb, (), m, D, acquired True ?take-sb O sb, release ?take-sb (dom S sb) R sb, S) →
  (is sb', θ sb, (), m, False, acquired True ?take-sb O sb, Map.empty, S).
from direct-computation.concurrent-step.Memop [OF i-bound' ts-i this]
have (ts, m, S) ⇒
  (ts i := (p sb, is sb', θ sb, (), False, acquired True ?take-sb O sb, Map.empty)], m, S).
moreover

have (ts sb', m sb, S sb) ~ (ts i := (p sb, is sb', θ sb, (), False, acquired True ?take-sb O sb, Map.empty)], m, S)
apply (rule sim-config.intros)
apply (simp add: ts sb' sb' O sb' R sb' S sb' m)

536
flush-all-until-volatile-nth-update-unused [OF i-bound ts_{sb-i})]

using share-all-until-volatile-write-Fence-commute

[OF i-bound ts_{sb-i} [simplified is_{sb} sb]]

apply (clarsimp simp add: S ts_{sb}=S_{sb} is_{sb} O_{sb} R_{sb} \s_b sb sb)

using leq

apply (simp add: ts_{sb}=ts_{sb})

using i-bound i-bound

apply (clarsimp simp add: Let-def nth-list-update

ts_{sb}=ts_{sb} sb sb sb sb sb sb O_{sb} R_{sb} S_{sb} D_{sb} \s_b sb sb sb sb)

done

ultimately

show ?thesis

using valid-own’ valid-hist’ valid-reads’ valid-sharing’ temps-distinct’ valid-sops’

valid-dd’ load-tempms-fresh’ enough-flushs’

valid-program-history’ valid’ m_{sb} S_{sb}’

by (auto simp del: fun-upd-apply)

next

case (SBHRMWRReadOnly 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: \text{¬} (cond ((sb_{sb}(t→m_{sb} a))) and

O_{sb} = O_{sb} and

R_{sb} = R_{sb} =Map.empty and

s_{sb}: s_{sb} = s_{sb}(t→m_{sb} a) and

D_{sb} = D_{sb} and

s_{sb}: s_{sb} = s_{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} \lor a \in \text{dom} S and

rels-cond: \forall j < \text{length ts}. i\neq j \rightarrow \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-locales)

show outstanding-non-volatile-refms-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-refms-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]

537
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 \((\ddot{\theta}_{sb}(t\mapsto m_{sb} a)) \ (\text{hd-prog } p_{sb} [])\) by simp
from valid-history-nth-update [OF i-bound this]
show \(?thesis\) by (simp add: \( ts_{sb}' sb' sb \ O_{sb}' \dot{\theta}_{sb}' \))
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 \( S_{sb}' ts_{sb}' \)
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]

538
show unowned-shared $S_{sb'}\, ts_{sb'}$ by (simp add: ts_{sb'} 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: $S_{sb'}\, ts_{sb'}\, sb$)

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 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'} \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 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': \( \forall \) sop\in\text{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-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'}

proof -

from data-dependency-consistent-instrs [OF i-bound ts_{sb}-i]

obtain

dd-is: data-dependency-consistent-instrs (dom \( \emptyset_{sb'} \)) is_{sb'}

by (auto simp add: is_{sb} \( \emptyset_{sb'} \))

from load-tmps-write-tmps-distinct [OF i-bound ts_{sb}-i]

have load-tmps is_{sb'} \cap \bigcup (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'})
have load-tmps-fresh': load-tmps-fresh ts\sb{b}'
proof -
from load-tmps-fresh [OF i-bound ts\sb{b}-i]
have load-tmps (RMW a t (D,f) cond ret A L R W# is\sb{b}') \cap dom \vartheta_{sb} = {} 
by (simp add: is\sb{a})
moreover
from load-tmps-distinct [OF i-bound ts\sb{b}-i] have t \notin load-tmps is\sb{a}'
by (auto simp add: is\sb{a})
ultimately have load-tmps is\sb{a}' \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{b}' sb\sb{b}' \vartheta_{sb}'
qed

from enough-flushs-nth-update [OF i-bound, where sb=[] ]
have enough-flushs': enough-flushs ts\sb{b}'
by (auto simp add: ts\sb{b}' sb\sb{b}'
have valid-program-history': valid-program-history ts\sb{b}'
proof -
have causal': causal-program-history is\sb{a}' sb'
by (simp add: is\sb{a} sb sb')
have last-prog p_{sb} sb' = p_{sb}
by (simp add: sb')
from valid-program-history-nth-update [OF i-bound causal' this]
show ?thesis by (simp add: ts\sb{b}' sb'')
qed

from is-sim have is: is = RMW a t (D,f) cond ret A L R W# is\sb{b}'
by (simp add: suspends sb is\sb{a})
with ts-i
have ts-i: ts!i = (p_{sb}, RMW a t (D,f) cond ret A L R W# is\sb{b}', \vartheta_{sb}(), D, acquired True ?take-sb O_{sb}, release ?take-sb (dom S_{sb}) R_{sb})
by (simp add: suspends sb)

have flush-all-until-volatile-write ts\sb{b} m_{sb} a = m_{sb} a
proof -
have \forall j < length ts_{sb}. i \neq j \rightarrow (let (\cdot,\cdot,\cdot,\cdot, sb_j,\cdot,\cdot,\cdot) = ts\sb{b}[j] 
in a \notin outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not \circ is-volatile-Write_{sb}) sb_j))
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 neq-i-j: i ≠ j
assume jth: ts_{sb\|j} = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)

have a /∈ outstanding-refs is-non-volatile-Write_{sb\|j} (takeWhile (Not ◦ is-volatile-Write_{sb\|j}) sb_j)

proof
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)

assume a-in: a ∈ outstanding-refs is-non-volatile-Write_{sb\|j} ?take-sb_j

with outstanding-refs-takeWhile [where P' = Not ◦ is-volatile-Write_{sb\|j}]

have a-in': a ∈ outstanding-refs is-non-volatile-Write_{sb\|j} sb_j

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 jth]]

have j-owns: a ∈ O_j \cup all-acquired sb_j

by auto


have no-unsharing:release ?take-sb_j (dom (S_{sb\|j})) R_j a ≠ Some False

by (auto simp add: Let-def)

from access-cond

show False

proof

assume a ∈ O_{sb\|j}

with ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb\|i} \cdot i \cdot jth]

j-owns

show False

by auto

next

assume a-shared: a ∈ dom S

with share-all-until-volatile-write-thread-local [OF ownership-distinct-ts_{sb\|j} sharing-consis-ts_{sb\|j} j-bound jth j-owns]

have a-dom: a ∈ dom (share ?take-sb_j S_{sb\|j})

by (auto simp add: S domIff)

from outstanding-non-volatile-writes-unshared [OF j-bound jth]

have non-volatile-writes-unshared S_{sb\|j} sb_j,

with non-volatile-writes-unshared-append [of S_{sb\|j} (takeWhile (Not ◦ is-volatile-Write_{sb\|j}) sb_j)]

have unshared-take: non-volatile-writes-unshared S_{sb\|j} (takeWhile (Not ◦ is-volatile-Write_{sb\|j}) 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

from flush-all-until-volatile-write-buffered-val-conv
[OF - i-bound ts\_sb-i this]
show \?thesis
by (simp add: sb)
qed

hence m\_a: m a = m\_sb a
by (simp add: m)

from cond have cond\:' \neg cond (\emptyset\_sb(t \mapsto m a))
by (simp add: m-a)

from direct-memop-step.RMWReadOnly [where cond=cond and \emptyset=\emptyset\_sb and m=m,
OF cond]
have (RMW a t (D, f) cond ret A L RW \# is\_sb ',
\emptyset\_sb , (,)\_sb, \_sb, R\_sb, S) \rightarrow
(is\_sb ', \emptyset\_sb (t \mapsto m a), (,)\_sb, m, False, \_sb, Map.empty, S).

from direct-computation.concurrent-step.Memop [OF i-bound\' ts-i [simplified sb,
simplified] this]
have (ts\_sb\' m, S) \Rightarrow_d (ts[i := (p\_sb, is\_sb\',
\emptyset\_sb (t \mapsto m a), (,)\_sb, False, \_sb, Map.empty)], m, S).

moreover

have tm\_sb-commute: \emptyset\_sb(t \mapsto (m\_sb a)) =
(\emptyset\_sb |' (dom \emptyset\_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, is\_sb\', \emptyset\_sb\' (t \mapsto m a), (,)\_sb, False, \_sb, Map.empty)], m, S)
apply (rule sim-config.intros)
apply (simp add: ts\_sb\' sb\' \_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 \_sb\' \emptyset\_sb\' sb\' sb)
using leq
apply (simp add: ts\_sb\')
using i-bound\' ts-sim
apply (clarsimp simp add: Let-def nth-list-update
ts\_sb\' sb\' sb\_sb\' \_sb\' R\_sb\' S\_sb\' \emptyset\_sb\' D\_sb\' \_sb\' ex-not m-a
split: if-split_asm)
apply (rule tm\_sb-commute)
done
ultimately
show \?thesis
using valid-own\' valid-hist\' valid-reads\' valid-sharing\' tm\_sb-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 (SBHRMWWrite cond t a D f ret A L R W)
then obtain
i_{sb}: i_{sb} = RMW a t (D,f) cond ret A L R W ≠ i_{sb}' and
cond: (cond (\(\theta_{sb}(t⇒m_{sb} a)\)) and
O_{sb}' : O_{sb} ∪ A – R and
R_{sb}' : R_{sb}' = Map.empty and
D_{sb}' : ¬ D_{sb}' and
\(\theta_{sb}' : \theta_{sb}' = \theta_{sb}(t⇒ret (m_{sb} a) (f (\(\theta_{sb}(t⇒m_{sb} a)\)))\) and
sb: sb=[ ] and
sb': sb'=[ ] and
m_{sb}' : m_{sb}' = m_{sb}(a := f (\(\theta_{sb}(t⇒m_{sb} a)\))) and
S_{sb}' : S_{sb}' = S_{sb} ⊕ W R ⊕ A L
by auto

from data-dependency-consistent-instrs [OF i-bound ts_{sb}-i]
have D-subset: D ⊆ dom \(\theta_{sb}\)
by (simp add: i_{sb})

from is-sim have is: is = RMW a t (D,f) cond ret A L R W ≠ i_{sb}'
by (simp add: suspends sb i_{sb})

with ts-i
have ts-i: ts!i = (p_{sb}, RMW a t (D,f) cond ret A L R W ≠ i_{sb}', \(\theta_{sb}(\cdot)\), D, O_{sb}, R_{sb})
by (simp add: suspends sb)

from safe-RMW-common [OF safe-memop-flush-sb [simplified i_{sb}]]
obtain access-cond: a ∈ O_{sb} ∨ a ∈ dom S and
rels-cond: ∀ j < length ts. i ≠ j −→ released (ts!j) a ≠ Some False
by (auto simp add: S sb)

have a-unflushed:
∀ j < length ts_{sb}. i ≠ j −→
(let (\(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot\)) = ts_{sb}[j]
\in a ≠ outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not o
is-volatile-Write_{sb}) sb_{j}))

proof –
{
fix j p_{j} is_{j} O_{j} R_{j} D_{j} x_{j} sb_{j}
assume j-bound: j < length ts_{sb}
assume neq-i-j: i ≠ j
assume jth: ts_{sb}[j] = (p_{j},is_{j}, x_{j}, sb_{j}, D_{j}, O_{j}, R_{j})
have a ≠ outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not o
is-volatile-Write_{sb}) sb_{j})
proof
let ?take-sb_{j} = (takeWhile (Not o is-volatile-Write_{sb}) sb_{j})
let ?drop-sb_j = (dropWhile (Not ◦ is-volatile-Write_{sb}) ?take-sb_j

assume a-in: a ∈ outstanding-refs is-non-volatile-Write_{sb} ?take-sb_j

with outstanding-refs-takeWhile [where P' = Not ◦ is-volatile-Write_{sb}]

have a-in': a ∈ outstanding-refs is-non-volatile-Write_{sb} sb_j

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 jth]]

have j-owns: a ∈ O_j ∪ all-acquired sb_j

by auto

with ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i jth]

have a-not-owns: a /∈ O_{sb} ∪ all-acquired sb

by blast

assume a-in: a ∈ outstanding-refs is-non-volatile-Write_{sb}

(takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j)

with outstanding-refs-takeWhile [where P' = Not ◦ is-volatile-Write_{sb}]

have a-in': a ∈ outstanding-refs is-non-volatile-Write_{sb} sb_j

by auto


have no-unsharing:release ?take-sb_j (dom (S_{sb})) R_j a ≠ Some False

by (auto simp add: Let-def)

from access-cond

show False

proof

assume a ∈ 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 ∈ 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 ∈ 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 ◦
is-volatile-Write_{sb}) sb_j)
(dropWhile (Not ◦ is-volatile-Write_{sb}) sb_j)]

have unshared-take: non-volatile-writes-unshared S_{sb} (takeWhile (Not ◦
is-volatile-Write_{sb}) 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: \ts_{sb})
qed

hence \m-a: \m a = m_{sb} a
by (simp add: \ts_{sb})

from cond have cond': cond (\theta_{sb}(t \mapsto m a))
by (simp add: \ts_{sb})

from safe-memop-flush-sb [simplified is_{sb}] cond'
obtain
L-subset: L \subseteq A and
A-shared-owned: A \subseteq \dom \s \cup O_{sb} and
R-owned: R \subseteq O_{sb} and
A-R: A \cap R = {} and
a-unowned-others-ts:
\forall j<\length \ts. i \neq j \longrightarrow (a \notin \owned (\ts!j) \cup \dom (\released (\ts!j))) and
A-unowned-by-others-ts:
\forall j<\length \ts. i \neq j \longrightarrow (A \cap (\owned (\ts!j) \cup \dom (\released (\ts!j)))) = {} and
a-not-ro: a \notin \read-only S
by cases (auto simp add: \ts_{sb})

from a-unowned-others-ts \ts\-sim leq
have a-unowned-others:
\forall j<\length \ts_{sb}. i \neq j \longrightarrow
(let (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \ts_{sb}\!’j in
a \notin \acquired \True (\takeWhile (\Not \circ \is-volatile-Write_{sb}) \ts_{sb}) \O_j \land
a \notin \all-shared (\takeWhile (\Not \circ \is-volatile-Write_{sb}) \ts_{sb})
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<\length \ts_{sb}. i \neq j \longrightarrow
(let (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \ts_{sb}\!’j in
in A \cap (\acquired \True (\takeWhile (\Not \circ \is-volatile-Write_{sb}) \ts_{sb}) \O_j \land
\all-shared (\takeWhile (\Not \circ \is-volatile-Write_{sb}) \ts_{sb})) = {})

545
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 /∈ read-only $S_{sb}$
proof
assume a: a ∈ 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 ∈ 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 $R_j$ $D_{sbj}$ $\emptyset_j$ sbj
assume j-bound: j < length $ts_{sb}$
assume $ts_{sb}$-j: $ts_{sb}$[j]=(p j,is $sbj$,$\emptyset_j$,$sbj$,$D_{sbj}$,O j,$R_j$)
assume neq-i-j: i≠j
have a /∈ unforwarded-non-volatile-reads (dropWhile (Not ◦ is-volatile-Write $sb$) sbj) {}
proof
let ?take-sb j = takeWhile (Not ◦ is-volatile-Write $sb$) sbj
let ?drop-sb j = dropWhile (Not ◦ is-volatile-Write $sb$) sbj
assume a-in: a ∈ unforwarded-non-volatile-reads ?drop-sb j {}

from a-unowned-others [rule-format, OF - neq-i-j] $ts_{sb}$-j j-bound
obtain a-unacq-take: a /∈ acquired True ?take-sb j O j and a-not-shared: a /∈ all-shared ?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]
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 .

note consis-j = sharing-consis [OF j-bound $ts_{sb}$-j]
with sharing-consis-append [of $S_{sb}$ O j ?take-sb j ?drop-sb j]
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

546
by auto

from in-unforwarded-non-volatile-reads-non-volatile-Readsb [OF a-in]
have a-in': a ∈ outstanding-refs is-non-volatile-Reads "drop-sb".

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" O_j) (flush "take-sb" m_{sb}) "drop-sb".

from read-only-share-all-shared [of a "take-sb" S_{sb}] a-not-ro' a-not-shared
have a-not-ro-j: a /∈ read-only (share "take-sb" 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" 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_j | (dom θ_j - read-tmps suspends_j),()),
  D_j, acquired True "take-sb" O_j, release "take-sb" (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, θ_j | (dom θ_j - read-tmps suspends_j),()),
  D_j, acquired True "take-sb" O_j, release "take-sb" (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 < 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
proof
  assume 3 sop a v ys zs A L R W.
  "drop-sb" = ys @ Write_{sb} True a' sop v A L R W # zs ∧ a ∈ A ∧
  a /∈ outstanding-refs is-Write_{sb} ys ∧ a' ≠ a
  with suspends_j
  obtain a' sop' v' ys zs A' L' R' W' where
  split-suspends: suspends_j = ys @ Write_{sb} True a' sop' v' A' L' R' W' # zs' (is
  suspends_j = ?suspends)
  and

547
a-A': a ∈ A' and
no-write: a /∈ outstanding-refs is-Write_{sb} (ys @ [Write_{sb} 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_{sb-j}]
have sharing-consis-j: Sharing-consistent S_{sb} O_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 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 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-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-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@[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}]

548
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
valid-sops-drop: \( \forall \text{sop} \in \text{write-sops} \ (\text{ys}@[\text{Write}_{\text{sb}} \ True \ \text{a'} \ \text{sop'} \ \text{v'} \ \text{A'} \ \text{L'} \ \text{R'} \ \text{W'}]) \). 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)
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: distinct-read-tmps-append)
done

from valid-history [OF j-bound ts_{sb-j}]
have h-consis:
history-consistent θ_j (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)
qued

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
h-consis] last-prog-hd-prog
have hist-consis': history-consistent θ_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@[Write_{sb} \ True \ a'} \ sop' \ v' \ A' \ L' \ R' \ W')) = {}

549
by (auto simp add: outstanding-refs-append suspends \_j \ [symmetric] split-suspends \_j )

have \text{acq-simp}:
\begin{align*}
& \text{acquired True } (ys @ [Write_{ab} True a' sop' v' A' L' R' W']) \\
& \quad \text{(acquired True } ?\text{take-sb}_j \mathcal{O}_j) = \\
& \quad \text{acquired True } ys \text{ (acquired True } ?\text{take-sb}_j \mathcal{O}_j) \cup A' - R' \\
& \text{by (simp add: acquired-append)}
\end{align*}

from flush-store-buffer-append \[\text{where} \ sb = ys@[Write_{ab} True a' sop' v' A' L' R' W'] \]
and \(sb' = zs'\), simplified,
\begin{align*}
& \text{OF j-bound-ts' is}_j \ [\text{simplified split-suspends}_j] \ \text{cph \ [simplified suspends}_j \\
& \text{ts'}_j \ [\text{simplified split-suspends}_j] \ \text{refl lp \ [simplified split-suspends}_j \ \text{reads-consis-ys} \\
& \text{hist-consis'} \ [\text{simplified split-suspends}_j \ \text{valid-sops-drop} \\
& \text{distinct-read-tmps-drop \ [simplified split-suspends}_j \ \text{no-volatile-Read}_{ab}\text{-volatile-reads-consistent} \ [\text{OF no-vol-read}, \text{where} \\
& \ \text{S} = \text{share} \ ?\text{drop-sb} S] \]
\end{align*}

obtain \(is'_j \mathcal{R}_j' \ where \)
\begin{align*}
& \text{is'}_j \ \text{instrs } zs' @ is'_{abj} = is'_j @ \text{prog-intrs} zs' \ \text{and} \\
& \text{steps-ys': (?ts', flush } ?\text{drop-sb} m, \ \text{share } ?\text{drop-sb} S) \ \Rightarrow d^* \\
& (?ts'_j | i = \text{last-prog} \\
& (hd-prog p_j (Write_{ab} True a' sop' v' A' L' R' W'\# zs')) (ys@[Write_{ab} True a' sop' v' A' L' R' W']) \\
& \text{is}_j' \ \\
& \hat{o}_j | i \ \text{dom } \hat{o}_j - \text{read-tmps} zs', \\
& (), \text{True, acquired True } ys \text{ (acquired True } ?\text{take-sb}_j \mathcal{O}_j) \cup A' - R', \mathcal{R}_j')], \\
& \text{flush } (ys@[Write_{ab} True a' sop' v' A' L' R' W']) \ (\text{flush } ?\text{drop-sb} m), \\
& \text{share } (ys@[Write_{ab} True a' sop' v' A' L' R' W']) \ (\text{share } ?\text{drop-sb} S)) \\
& (is (\cdot,\cdot,\cdot) \Rightarrow d^* (?ts-ys, ?m-ys, ?shared-ys)) \\
& \text{by (auto simp add: acquired-append outstanding-refs-append)}
\end{align*}

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!l = (p_{ab}, is_{ab}, \hat{o}_{ab},()), \\
\mathcal{D}_{ab}, \text{acquired True } sb \ \mathcal{O}_{ab}, \ \text{release } sb \ \text{(dom } S_{ab} \ \mathcal{R}_{ab}) \\
\text{by simp} \\
\text{note} \ \text{conflict-computation} = \text{rtranclp-trans} \ [\text{OF steps-flush-sb steps-ys}] \\
\end{align*}

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation]

from flush-unchanged-addresses [OF no-write]
have flush (ys @ [Write_{ab} True a' sop' v' A' L' R' W']) m a = m a. \\
with safe-delayedE [OF safe i-bound-ys ts-ys-i, simplified is_{ab}] cond'
have a-unowned:
∀ j < length ?ts-ys. i\neq j \rightarrow (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 leq A'R'
show False
by (auto simp add: Let-def)
next
assume \exists A L R W ys zs. ?drop-sb\_j = ys @ Ghost\_sb A L R W# zs \land a \in A \land a \notin 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 \in A' and
no-write: a \notin 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 O_j sb\_j,
then have A'R': 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 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\_j \rangle
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)

551
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-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\_j reads-consistent-append)

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@[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 (?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 \_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

552
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\(^t\): history-consistent \(\vartheta_j\) (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\(_{ab}\)
\(\mbox{(ys}@[\text{Ghost}_{ab} A' L' R' W']) = \{\}\)
by (auto simp add: outstanding-refs-append suspends \(j\) [symmetric]
split-suspends \(j\))

have acq-simp:
acquired True \(\mbox{(ys}@[\text{Ghost}_{ab} A' L' R' W'])\)
\(\mbox{(acquired True \(?\text{take-sb}_j\) O\(_j\)) = \mbox{acquired True \mbox{ys} (acquired True \(?\text{take-sb}_j\) O\(_j\)) \cup A' - R'}\)
by (simp add: acquired-append)

from flush-store-buffer-append [where \(sb=\mbox{ys}@[\text{Ghost}_{ab} A' L' R' W']\) 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=\mbox{share \(?\text{drop-sb}\ S\)}\)

obtain is\(_j'\) R\(_j'\) where
\(\mbox{is\(_j'\): instrs \mbox{zs}@[\text{Ghost}_{ab} A'] = \mbox{is\(_j'\): instrs zs'}\) and
steps-ys: \(?\mbox{ts}', \mbox{flush \(?\text{drop-sb m, share \(?\text{drop-sb S)}\)}\) \Rightarrow d^\ast\)
\(?\mbox{ts}'[j:=\mbox{(last-prog}}
\langle \mbox{hd-prog} \(p_j\) (\text{Ghost}_{ab} A' L' R' W'\# zs') \rangle \rangle \rangle \left(\mbox{ys}@[\text{Ghost}_{ab} A' L' R' W']\right),
\(\mbox{is\(_j'\)},\)
\(\mbox{\vartheta_j}\mid (\text{dom} \vartheta_j - \text{read-tmps} zs'),\)
\(\emptyset\),
\(D_j \vee \mbox{outstanding-refs is-volatile-Write}_{ab}\ \mbox{ys} \neq \{\}, \mbox{acquired True \mbox{ys}
\mbox{(acquired True \(?\text{take-sb}_j\) O\(_j\)) \cup A' - R',R\(_j'\)}]}\),
\mbox{flush \mbox{(ys}@[\text{Ghost}_{ab} A' L' R' W']) \mbox{(flush \(?\text{drop-sb m)}\),}
\mbox{share \mbox{(ys}@[\text{Ghost}_{ab} A' L' R' W']) \mbox{(share \(?\text{drop-sb S)}\))}
\mbox{(is \((-,-,\cdot) \Rightarrow d^\ast\ (\text{?ts-ys,?m-ys,?
\text{shared-ys}}))}}\)
by (auto simp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: i < length \(?\mbox{ts-ys}
by auto

553
from i-bound\' neq-i-j
have ts-ys-i: \(\exists ts, ys. i = (p_{sb}, is_{sb}, \emptyset_{sb}).(\) \\
\(\mathcal{D}_{sb}, acquired\ True sb \mathcal{O}_{sb}, release sb (dom \mathcal{S}_{sb} \mathcal{R}_{sb})\) \\
by simp
note conflict-computation = rtranclp-trans [OF steps-flush-sb steps-ys]

from safe-reach-safe-rtrancl [OF safe-reach conflict-computation] 

from flush-unchanged-addresses [OF no-write] 
have flush (ys @ [\text{Ghost}_{sb} A^' L^' R^' W^']) m a = m a.

with safe-delayedE [OF safe i-bound-ys ts-ys-i, simplified is_{sb}] cond\' 
have a-unowned:

\[\forall j < \text{length } ts-ys. i \neq j \rightarrow (let (O_j) = \text{map owned } ts-ys!j \text{ in } a \notin O_j)\]
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 leq A^' R'

show False 
by (auto simp add: Let-def)
qed
then show False
by simp
qed

have A-unused-by-others:
\[\forall j < \text{length } (map \mathcal{O}_{sb} ts_{sb}). i \neq j \rightarrow\]
\[\text{(let } (O_j, sb_j) = \text{map } \mathcal{O}_{sb} ts_{sb}! j \text{ in } A \cap (O_j \cup \text{outstanding-refs is-volatile-Write}_{sb} sb_j) = \{\})\]
proof – 
{
fix j O_j sb_j 
assume j-bound: j < \text{length } (map owned ts_{sb}) 
assume neq-i-j: i \neq j 
assume ts_{sb}-j: (map \mathcal{O}_{sb} ts_{sb})!j = (O_j, sb_j) 
assume conflict: A \cap (O_j \cup \text{outstanding-refs is-volatile-Write}_{sb} sb_j) \neq \{\}
have False 
proof – 
from j-bound leq 
have j-bound\': j < \text{length } (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

554
by simp

from conflict obtain a' where
  a-in: a' ∈ A and
  conflict: a' ∈ Oj ∨ a' ∈ outstanding-refs is-volatile-Write sbj sbj
by auto
  from A-unowned-by-others [rule-format, OF - neq-i-j] j-bound tsjb-j
  have A-unshared-j: A ∩ all-shared (takeWhile (Not o is-volatile-Write sbj) sbj) = {}
  by (auto simp add: Let-def)
from conflict
show thesis
proof

assume a' ∈ Oj

from all-shared-acquired-in [OF this] A-unshared-j a-in
have conflict: a' ∈ acquired True (takeWhile (Not o is-volatile-Write sbj) sbj) Oj
by (auto)
  with A-unowned-by-others [rule-format, OF - neq-i-j] j-bound tsjb-j a-in
  show False by auto
next
assume a-in-j: a' ∈ outstanding-refs is-volatile-Write sbj sbj

  let ?take-sb j = (takeWhile (Not o is-volatile-Write sbj) sbj)
  let ?drop-sb j = (dropWhile (Not o is-volatile-Write sbj) sbj)

  from ts-sim [rule-format, OF j-bound''] tsjb-j j-bound''
  obtain pjb suspendsjb isjb Djb Rj θsbjb isjb where
  tsjb-j: tsjb ! j = (pjb,isjb, θsbjb, sbj,Dsbj,Ωj,Rj) and
  suspendsjb: suspendsjb = ?drop-sb j and
  Djb: Dsbj = (Dj ∨ outstanding-refs is-volatile-Write sbj sbj ≠ {}) and
  isjb: instrs suspendsjb @ isjb = isj @ prog-instrs suspendsj and
  tsj: tsj = (hd-prog pj suspendsj, isj,
  θsbjb | (dom θsbjb - read-tmps suspendsj),(), Dj), acquired True ?take-sb j Oj,release
  ?take-sb j (dom Sjb) Rj)
  apply (cases tsjb')
  apply (force simp add: Let-def)
done

have a' ∈ outstanding-refs is-volatile-Write sbj suspendsj
proof
from a-in-j
have a' ∈ outstanding-refs is-volatile-Write sbj (?take-sb j @ ?drop-sb j)
  by simp
thus thesis
apply (simp only: outstanding-refs-append suspendsj)
apply (auto simp add: outstanding-refs-conv dest: set-takeWhileD)
from split-volatile-Write\textsubscript{sb}-in-outstanding-refs [OF this]

obtain sop' \cdot ys zs A' L' R' W' where
split-suspends\textsubscript{j}: suspends\textsubscript{j} = ys @ Write\textsubscript{sb} \ True a' sop' \cdot A' L' R' W'\# zs (is suspends\textsubscript{j} = ?suspends)
by blast

from valid-program-history [OF j-bound'' ts\textsubscript{sb}-j]
have causal-program-history is\textsubscript{sbj} sb\textsubscript{j}.
then have cph: causal-program-history is\textsubscript{sbj} ?suspends
apply –
apply (rule causal-program-history-suffix [where sb=?take-sb\textsubscript{j} ] )
apply (simp only: split-suspends\textsubscript{j} [symmetric] suspends\textsubscript{j})
apply (simp add: split-suspends\textsubscript{j})
done

from valid-last-prog [OF j-bound'' ts\textsubscript{sb}-j] have last-prog: last-prog p\textsubscript{j} sb\textsubscript{j} = p\textsubscript{j}.
then have lp: last-prog p\textsubscript{j} ?suspends = p\textsubscript{j}
apply –
apply (rule last-prog-same-append [where sb=?take-sb\textsubscript{j}])
apply (simp only: split-suspends\textsubscript{j} [symmetric] suspends\textsubscript{j})
apply simp
done

from valid-reads [OF j-bound'' ts\textsubscript{sb}-j]
have reads-consis: reads-consistent False \textsubscript{O}\textsubscript{j} m\textsubscript{sb} sb\textsubscript{j}.
from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing S\textsubscript{sb} ts\textsubscript{sb} j-bound''

\textsubscript{ts\textsubscript{sb}-j this]}
have reads-consis-m-j: reads-consistent True (acquired True ?take-sb\textsubscript{j} O\textsubscript{j}) m suspends\textsubscript{j}
by (simp add: m suspends\textsubscript{j})

from outstanding-non-volatile-refs-owned-or-read-only [OF j-bound'' ts\textsubscript{sb}-j]
have nvo-j: non-volatile-owned-or-read-only False S\textsubscript{sb} O\textsubscript{j} sb\textsubscript{j},
with non-volatile-owned-or-read-only-append [of False S\textsubscript{sb} O\textsubscript{j} ?take-sb\textsubscript{j} ?drop-sb\textsubscript{j}]
have nvo-take-j: non-volatile-owned-or-read-only False S\textsubscript{sb} O\textsubscript{j} ?take-sb\textsubscript{j}
by auto

from a-unowned-others [rule-format, OF - neq-i-j] ts\textsubscript{sb}-j j-bound
have a-not-acq: a \notin acquired True ?take-sb\textsubscript{j} O\textsubscript{j}
by auto

from a-notin-unforwarded-non-volatile-reads-drop[OF j-bound'' ts\textsubscript{sb}-j neq-i-j]
have a-notin-unforwarded-reads: a \notin unforwarded-non-volatile-reads suspends\textsubscript{j} {}
by (simp add: suspends_j)

let ?ma=(m(a := f (θ_ab(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_ O_j) ?ma suspends_j
by auto

from reads-consis-ma-j
have reads-consis-ys: reads-consistent True (acquired True ?take-sb_ O_j) ?ma (ys)
by (simp add: split-suspends_j reads-consistent-append)

from direct-memop-step.RMWWrite [where cond=cond and θ=θ_ab and m=m, OF cond’]
have (RMW a t (D, f) cond ret A L R W# i_{sb}', θ_{sb}, (), m,D, O_{sb}, R_{sb}, S) →
(is_{sb}', θ_{sb}(t → ret (m a) (f (θ_{ab}(t → m a)))), (), ?ma, False, O_{sb} ∪ A - R, Map.empty, S ⊕ W R ⊖ A L).
from direct-computation.concurrent-step.Memop [OF i-bound’ ts-i this]
have step-a: (ts, m, S) ⇒_d
(ts[i := (psb, i_{sb}', θ_{sb}(t → ret (m a) (f (θ_{ab}(t → m a)))), (), False, O_{sb} ∪ A - R,Map.empty)],
?ma,S ⊕ W R ⊖ 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,
θ_{sbj} |' (dom θ_{sbj} - read-tmps suspends_j),(), D_j, acquired True ?take-sb_j O_j,release
?take-sb_j (dom (Δ_{sb}) R_j)
by auto

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

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

557
apply (simp only: split-suspendsj [symmetric] suspendsj)
apply (simp only: distinct-read-tmps-append)
done

from valid-history [OF j-bound" ts_{sb-j}]
have h-consis:
  history-consistent \( \sigma_{sbj} \) (hd-prog \( p_j \) (?take-sb\( j \) @suspends\( j \)) (?take-sb\( j \) @suspends\( j \))
apply (simp only: split-suspendsj [symmetric] suspendsj)
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-suspendsj [symmetric] suspendsj)
  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-suspendsj [symmetric] suspendsj)
  by (rule last-prog-hd-prog-append)
  ultimately show ?thesis
  by (simp add: split-suspendsj [symmetric] suspendsj)
qed

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop
h-consis] last-prog-hd-prog
have hist-consis': history-consistent \( \sigma_{sbj} \) (hd-prog \( p_j \) suspends\( j \)) suspends\( j \)
  by (simp add: split-suspendsj [symmetric] suspendsj)
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-suspendsj)
from j-bound' have j-bound-ts-a: \( j < \text{length} \ ?ts-a \) 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-suspendsj] cph [simplified suspendsj]
hist-consis' [simplified split-suspendsj] valid-sops-drop
  distinct-read-tmps-drop [simplified split-suspendsj]
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) ⇒ \(d^*\)

\((?ts-a[j] := \text{last-prog} (\text{hd-prog} p_j zs) ys, \)

\(\text{is}_j', \)

\(\vartheta_{sbj} \mid (\text{dom} \vartheta_{sbj} - \text{read-tmps} zs), \)

\(((), D_j \lor \text{outstanding-refs} \text{is-volatile-Write}_{sb} \) \(ys \neq \{\}, \text{acquired True} \)

\(ys \) \((\text{acquired True} ?\text{take-sb}_{j} O_{j}).R_{j}')]], \)

\(\text{flush} \) \((ys (?ma), \_ \text{share} \) \((ys (?shared-a)) \)

\((\text{is} ('-, -, -) ⇒ \(d^*\) (?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: 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 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}, is_{sb}'', 
\(\vartheta_{sb}(t \mapsto \text{ret} (m a)) \(f (\vartheta_{sb}(t \mapsto \text{m a}))),(), \text{False}, O_{sb} \cup \text{A} - \text{R,Map.empty}) \)
by simp

from j-bound-ts-a have j-bound-ys: j < length ?ts-ys
by auto
then have ts-ys-j: ?ts-yslj = (last-prog (hd-prog p_j zs) ys, Write True a' sop' A' L' R' W'#is_{j}'', \(\vartheta_{sbj} \mid (\text{dom} \vartheta_{sbj} - \text{read-tmps} zs), ()\), D_j \lor \text{outstanding-refs} \text{is-volatile-Write}_{sb} \) \(ys \neq \{\}, \text{acquired True} \)
ys \((\text{acquired True} ?\text{take-sb}_{j} O_{j}).R_{j}') \)
by (clarsimp simp add: is_{j}')

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:
\(\forall i < \text{length ts}. j \neq i \longrightarrow (let (O_i) = \text{map owned} \) ?ts-ysli 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' A-R
  show False
by (auto simp add: Let-def)
qed
qed
}
thus ?thesis
by (auto simp add: Let-def)
qed

have A-unacquired-by-others:
  ∀j<length (map O-sb ts_s_b). i ≠ j →
    (let (O_j, sb_j) = map O-sb ts_s_b! j
     in A ∩ all-acquired sb_j = {})
proof -
{
  fix j O_j sb_j
  assume j-bound: j < length (map owned ts_s_b)
  assume neq-i-j: i ≠ j
  assume ts_s_b-j: (map O-sb ts_s_b)!j = (O_j, sb_j)
  assume conflict: A ∩ all-acquired 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_s_b
      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 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_s_b-j j-bound'''
    obtain p_j suspends_s_j is_s_j sb_j D_s_bj R_j D_j is_j where
      ts_s_b-j: ts_s_b! j = (p_j,is_s_j,D_s_bj, R_j,D_j, is_j)
      and
      suspends_s_j: suspends_s_j = ?drop-sb_j and
      is_j: instrs suspends_s_j @ is_s_j = is_j @ prog-instrs suspends_s_j and
      D_j: D_s_bj = (D_j ∨ outstanding-refs is-volatile-Write_s_b sb_j ≠ {} ) and
      ts_j: ts!j = (hd-prog p_j suspends_s_j, is_j,
       θ_s_bj |' (dom θ_s_bj - read-tmps suspends_s_j),(),
       D_j, acquired True ?take-sb_j O_j,release ?take-sb_j (dom S_s_b) R_j)
    apply (cases ts_s_b!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 ?thesis
proof
  assume ∃ sop a′′ v ys zs A L R W.
suspends\_j = ys @ Write\_sb\_j True a′′ sop v A L R W# zs ∧ a′ ∈ A
then
obtain a′′ sop′ v′ y s z A′ L′ R′ W′ where
  split-suspends\_j: suspends\_j = ys @ Write\_sb\_j True a′′ sop′ v′ A’ L’ R’ W’# zs (is suspends\_j)
  = ?suspends) and
  a′'-A!: a′ ∈ A'
  by blast

from valid-program-history [OF j-bound′′ ts\_sb\_j]
have causal-program-history is\_sb\_j 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 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\_j sb\_j.
from reads-consistent-flush-all-until-volatile-write [OF
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-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. 
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 (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 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 \===0sb and m=m, OF cond] 
have (RMW a t (D, f) cond ret A L R W# i sb’, \ 
\sb, (\), m, D, O sb, R sb, S) \rightarrow 
(i sb’, \ 
\sb(t ↦ ret (m a) (f (\sb (t ↦ m a)))), (\), ?ma, False, O sb \cup A − R,Map.empty, S ⊕ 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’, \sb (t ↦ ret (m a) (f (\sb (t ↦ m a)))), (\), False, O sb \cup A − R,Map.empty)], \ 
?ma,S \oplus W R \ominus A L) 
(is - \Rightarrow d (?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,
\(\partial_{sbj} \upharpoonright \ (\text{dom } S_{sb}) - \text{read-tmps suspends}_j \)),
\(D_j\), acquired True \(?\text{take-sb}_j \in \theta_{sb}\), release \(?\text{take-sb}_j\) (dom \(S_{sb}\) \(\mathcal{R}_j\))
by auto

from valid-write-sops [OF j-bound'' ts\_sb-j]
have \(\forall \text{sop}\in\text{write-sops} \ (?\text{take-sb}_j \in \text{dom } S_{sb})\). 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: \(\forall \text{sop}\in\text{write-sops} (ys)\). 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 (?\text{take-sb}_j \in \text{dom } S_{sb})
by (simp add: split-suspends\_j [symmetric] suspends\_j)
then obtain
read-tmps-take-drop: \(\forall \text{sop}\in\text{write-sops} \ ?\text{take-sb}_j\). valid-sop sop
and valid-sops-drop: \(\forall \text{sop}\in\text{write-sops} (ys)\). valid-sop sop
apply (simp only: write-sops-append)
apply auto
done

from valid-history [OF j-bound'' ts\_sb-j]
have h-consis:
history-consistent \(\theta_{sbj} \ (\text{hd-prog p}\_j \ (?\text{take-sb}_j \in \text{dom } S_{sb})\) ) (?\text{take-sb}_j \in \text{dom } S_{sb})
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 \in ?\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 \in \text{dom } S_{sb})\)\ ?\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 \(?\text{thesis}\)
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 \( \theta \) _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]
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\( j \)]
  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\( _{sb} \), vol-
  write-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_ \( | (\text{dom} \theta \) _sbj_ \) - read-tmps zs),
    ()),
    \( D_j \lor \) outstanding-refs is-volatile-Write\( _{sb} \) ys \( \neq \) {}, acquired True ys (acquired
    True ?take-sb\( j \) 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

564
by auto

from i-bound' neq-i-j 
have ts-ys-i: ?ts-ys!i = (p sb', is sb', 
  \( (t \mapsto \text{ret } (m \ a)) \), False, \( O_{sb} \cup A - R, \text{Map.empty} \) 
) 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'' , 
  \( \vartheta_{sb} (t \mapsto \text{read-tmps } zs) , () , \) , 
  \( D_{j} \cup \text{outstanding-refs } is\text{-volatile-Write}_{sb} \text{ ys } \neq \{\} , \) 
  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-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: 
\( \forall i < \text{length } ?\text{ts-ys}, j \neq i \longrightarrow \langle \text{let } (O_i) = \text{map owned } ?\text{ts-ys!i in } A' \cap O_i = \{\} \rangle \) 
apply cases 
apply (clarsimp 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 \( \exists A L R W ys zs. \text{suspends}_j = ys @ \text{Ghost}_{sb} A L R W# zs \land a' \in A \) 
  then 
  obtain ys zs A' L' R' W' where 
    split-suspends_j: \text{suspends}_j = ys @ \text{Ghost}_{sb} A' L' R' W'# zs (is \text{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 sbj. 
then have cph: causal-program-history is sbj ?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 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 \(\Rightarrow\) suspends = p_j
  apply (rule last-prog-same-append [where sb=?take-sb_j])
  apply (simp only: split-suspends \(\Rightarrow\) symmetric suspends)
  apply simp
  done

from valid-reads [OF j-bound'' ts_{sb-j}]
have reads-consis: reads-consistent False \(\mathcal{O}_j\) m_{sb} sb_j.
from reads-consistent-flush-all-until-volatile-write \(\langle\text{valid-ownership-and-sharing} \mathcal{S}_{sb} ts_{sb}\rangle\text{ j-bound''}
  ts_{sb\cdot j} \text{ this}\)
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 \(\mathcal{S}_{sb} \mathcal{O}_j\) sb_j.
with non-volatile-owned-or-read-only-append [of False \(\mathcal{S}_{sb} \mathcal{O}_j\) ?take-sb_j ?drop-sb_j]
have nvo-take-j: non-volatile-owned-or-read-only False \(\mathcal{S}_{sb} \mathcal{O}_j\) ?take-sb_j
  by auto

from a-unowned-others [rule-format, OF - neq-i-j] ts_{sb\cdot j} j-bound
have a-not-acq: a \(\notin\) acquired True ?take-sb_j \(\mathcal{O}_j\)
  by auto

from a-notin-unforwarded-non-volatile-reads-drop[OF j-bound'' ts_{sb\cdot j} neq-i-j]
have a-notin-unforwarded-reads: a \(\notin\) 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 \(\mathcal{O}_j\)) ?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 \(\Rightarrow\) reads-consistent-append)

from direct-menop-step.RMWWrite [where cond=cond \(\Rightarrow\) \(\theta=\theta_{sb}\) and m=m, OF cond']
have (RMW a t (D, f) cond ret A L R W# \(\mathcal{S}_{sb}'\),
  \(\theta_{sb}\), (), m, \(\mathcal{D},\mathcal{O}_{sb}\), \(\mathcal{R}_{sb}\), \(\mathcal{S}\)) \(\Rightarrow\)
from direct-computation.concurrent-step.Memop [OF i-bound' ts-i [simplified sb, simplified] this]

have step-a: (ts, m, S) ⇒_d
  ([ts|i := (psb', isb', θsb(t ↦ ret (m a) (f (θsb(t ↦ ret (m a))))), (), False, Osb ∪ A - R, Map.empty, S ⊕ W R ⊖ A L)],
   ?ma, S ⊕ W R ⊖ A L)
  (is - ⇒_d (?ts-a, - ?shared-a)).

from ts_j neq-i-j j-bound

have ts-a-j: ?ts-a!j = (hd-prog pj suspendsj, isj,
  θsbj | (dom θsbj - read-tmps suspendsj), () , D_j, acquired True ?take-sb_j O_j, release
  ?take-sb_j (dom Ssb_j) R_j)
  by auto

from valid-write-sops [OF j-bound'' ts_j]

have ∀ sop∈write-sops (?take-sb_j@?suspends). valid-sop sop
  by (simp add: split-suspendsj [symmetric] suspendsj)
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_j]

have distinct-read-tmps (?take-sb_j@?suspendsj)
  by (simp add: split-suspendsj [symmetric] suspendsj)
then obtain read-tmps-take-drop: read-tmps ?take-sb_j ∩ 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'' ts_j]

have h-consis:
  history-consistent θsbj (hd-prog pj (?take-sb_j@?suspendsj)) (?take-sb_j@?suspendsj)
  apply (simp only: split-suspendsj [symmetric] suspendsj)
  apply simp
  done

have last-prog-hd-prog: last-prog (hd-prog pj sb_j) ?take-sb_j = (hd-prog pj suspendsj)
proof -

from last-prog have last-prog pj (?take-sb_j@?drop-sb_j) = pj
  by simp

567
from last-prog-hd-prog-append' [OF h-consis] this
have last-prog (hd-prog p j suspends j) ?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 (?take-sb @ suspends j)) ?take-sb =
   last-prog (hd-prog p j suspends j) ?take-sb
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'=Ghost sb A' L' R' W'#zs,
simplified]
   OF j-bound-ts-a 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-Readsb-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',
    \_sbj \_ (dom \_sbj - read-tmps zs),
    ()),
    D j \lor outstanding-refs is-volatile-Write sb ys \neq {}
  ), acquired True ys (acquired True ?take-sb q 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 @ [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 causal_l: causal-program-history is_{sbj} zs by (rule cph')

from causal_l, causal-program-history [of [], simplified, OF refl] is_{j}'
obtain is_{j}''
where is_{j}': is_{j}' = Ghost A'L'R'W'#is_{j}'' and
is_{j}'': instrs zs @ is_{sbj} = is_{j}'' @ prog-intrs 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, \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, Ghost A'L'R'W'#is_{j}''',
\vartheta_{sb} | \{ \text{dom } \vartheta_{sb} - \text{read-tmps zs} \}, ()),
\mathcal{D}_j \forall \text{outstanding-refs is-volatile-\text{Write}_{sb} ys} \neq \{\},
\text{acquired True ys (acquired True ?\text{take-sb} O_j, R_j)}
by (clarsimp simp add: is_{j}')
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 \{ \mathcal{O}_i \} = \text{map owned ?ts-ys!i in A' \cap } \mathcal{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
qed
}
thus ?thesis
by (auto simp add: Let-def)
qed

569
\begin{verbatim}
{ fix j
  fix p_j \is_{sbj} O_j R_j D_{sbj} \emptyset_j sb_j
  assume j-bound: j < length ts_{sb}
  assume ts_{sb}: j = (p_j, is_{sbj}, \emptyset_j, sb_j, D_{sbj}, O_j, R_j)
  assume neq-i-j: i \neq j
  have A \cap read-only-reads (acquired True (takeWhile (Not \is-volatile-Write sb) sb_j) O_j)
    (dropWhile (Not \is-volatile-Write sb) sb_j) = {} 
  proof -
  { let ?take-sb_j = (takeWhile (Not \is-volatile-Write sb) sb_j)
    let ?drop-sb_j = (dropWhile (Not \is-volatile-Write sb) sb_j)
    assume conflict: A \cap read-only-reads (acquired True ?take-sb_j O_j) ?drop-sb_j \neq {}
    have False
    proof -
      from conflict obtain a' where
      a'-in: a' \in A and
      a'-in-j: a' \in read-only-reads (acquired True ?take-sb_j O_j) ?drop-sb_j
      by auto
      from ts-sim [rule-format, OF j-bound] ts_{sb}: j-bound
      obtain p_j suspends_j is_{sbj} D_{sbj} \emptyset_{sbj} is_j where
      ts_{sb}: j = (p_j, is_{sbj}, \emptyset_{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 \lor \outstanding-refs is-volatile-Write sb_j \neq \{\}) and
      ts: ts[j] = (\hd-prog p_j suspends_j, is_j, \emptyset_{sbj} \lceil (dom \emptyset_{sbj}) \land \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 (clarsimp 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'-unacq: a' \notin acquired True ys (acquired True ?take-sb_j O_j)
      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} ?suspends
      proof -
      apply (rule causal-program-history-suffix [where sb=?take-sb_j])
      apply (simp only: split-suspends_j [symmetric] suspends_j)
      done
  }
}
\end{verbatim}
from valid-last-prog [OF j-bound ts_{sb\cdot 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 ts_{sb\cdot j}]
have reads-consis: reads-consistent False \mathcal{O}_j m_{sb \cdot j}.

from reads-consistent-flush-all-until-volatile-write [OF \{valid-ownership-and-sharing\} S_{sb \cdot j}

j-bound

\textit{ts}_{sb\cdot j} this]

have reads-consis-m-j: reads-consistent True (acquired True ?take-sb) \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\cdot j}]

have nvo-j: non-volatile-owned-or-read-only False S_{sb \cdot j} sb_j,

with non-volatile-owned-or-read-only-append [of False S_{sb \cdot j} ?take-sb \ ?drop-sb]

have nvo-take-j: non-volatile-owned-or-read-only False S_{sb \cdot j} ?take-sb_j

by auto

from a-unowned-others [rule-format, OF - neq-i-j] ts_{sb\cdot j} j-bound

have a-not-acq: a \notin \textit{acquired} True ?take-sb \ \mathcal{O}_j
by auto

from a-notin-unforwarded-non-volatile-reads-drop[OF j-bound ts_{sb\cdot j} neq-i-j]

have a-notin-unforwarded-reads: a \notin \textit{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 [\textit{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 \ \mathcal{O}_j) ?ma suspends_j
by auto

from reads-consis-ma-j

have reads-consis-ys: reads-consistent True (acquired True ?take-sb \ \mathcal{O}_j) ?ma (ys)
by (simp add: split-suspends suspends_j reads-consistent-append)

from direct-memop-step,RMWWrite [\textit{where} cond=cond and \partial=\partial_{sb} and m=m, OF cond’]

have (RMW a t (D, f) cond ret A L R W\# i_{sb’}, \partial_{sb}, (), m, \mathcal{D},\mathcal{O}_{sb},\mathcal{R}_{sb},\mathcal{S}) \rightarrow

(i_{sb’}, \partial_{sb}(t \mapsto ret (m a)) (f (\partial_{sb} (t \mapsto m a)))), (), \partial_{ma}, False, \mathcal{O}_{sb} \cup A −

R,Map.empty, \mathcal{S} \oplus_{W} R \oplus_{A} L).

571
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′, q sb(t ↦ ret (m a) (f (q sb(t ↦ m a)))), (), False, O sb ∪ A − R, Map.empty)],
  ?ma,S ⊕ wR ⊖ A L)
  (is - ⇒ d (?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,
  q sbj |' (dom q 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-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

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 = {}
    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 q 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 =

572
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]
have no-vol-read: outstanding-refs is-volatile-Read\_sb\_j (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\_j False a' 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\_j-volatile-reads-consistent [OF no-vol-read], where
\( S = \#shared-a \)

obtain is\_j' \( \mathcal{R}_j' \) where
is\_j': Read False a' t'# instrs zs @ is\_sb\_j = 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 |' (\text{dom } \theta \sbj - \text{insert } t' (\text{read-tmps zs})),
(), D_j \lor \text{outstanding-refs is-volatile-Write\_sb\_j ys } \neq \{\}, \text{acquired True ys (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 is\_sb\_j \((ys @ [Read\_sb\_j False a' t' v']) @ zs)\)
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')

573
from causal.causal-program-history [of [], simplified, OF refl] isj′

obtain isj′′

where isj′: isj′ = Read False a’ t’#isj′′ and

isj′′: instrs zs @ isj′ = isj′′ @ 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’#isj′′, 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-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’ ∈ 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’ ∈ O_j ∪ all-acquired sb_j

proof −

{ assume a-notin: a’ /∈ O_j ∪ 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 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 ⊆ 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_j [of sb_j O_j]
  all-acquired-append [of ?take-sb_j ?drop-sb_j]
have a' \notin acquired True (takeWhile (Not ◦ is-volatile-Write sb_j) 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 (S ⊕ W R ⊕ A L) .
with a'-not-ro have False
  by auto
with a-notin read-only-share-unowned [OF weak-consis - a'-ro]
  all-acquired-takeWhile [of (Not ◦ is-volatile-Write sb_j) sb_j]
have a' \in read-only (S ⊕ W R ⊕ A L)
  by (auto simp add: acquired-takeWhile-non-volatile-Write sb_j)
with a'-not-ro have False
  by auto
}
thus ?thesis by blast
qed

moreover
from A-unacquired-by-others [rule-format, OF - neq-i-j] ts_{sb_j} j-bound
have A \cap all-acquired sb_j = {}
by (auto simp add: Let-def)
mmoreover
from A-unowned-by-others [rule-format, OF - neq-i-j] ts_{sb_j} j-bound
have A \cap O_j = {}
  by (auto simp add: Let-def dest: all-shared-acquired-in)
mmoreover note a'-in
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 \( O_j \) \( R_j \) \( D_j \) \( \theta_j \) \( sb_j \) \( p_j \)
assume j-bound: \( j < \) length \( ts_{sb}' \)
assume \( ts_{sb}' j \): \( ts_{sb}' j = (p_j, is_j, \theta_j, sb_j, O_j, R_j) \)
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: \( ts_{sb}' \) \( sb \) \( sb' \))
next
case False
from j-bound have j-bound': \( j < \) length \( ts_{sb} \)
by (auto simp add: \( ts_{sb}' \))
with \( ts_{sb}' j \) False i-bound
have \( ts_{sb}' j \): \( ts_{sb}' j = (p_j, is_j, \theta_j, sb_j, O_j, R_j) \)
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 A-no-read-only-reads [OF j-bound' \( ts_{sb}' j \) False [symmetric]] L-subset
have \( \forall \) a\( \in \)read-only-reads
\((\text{acquired True (takeWhile (Not \circ is-volatile-Write}_{sb}) sb_j) O_j))\)
\((\text{dropWhile (Not \circ is-volatile-Write}_{sb}) sb_j).)\)
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
next
show outstanding-volatile-writes-unowned-by-others \( ts_{sb}' \)
proof (unfold-locales)
fix \( i_1 j p_1 is_1 O_1 R_1 D_1 x_1 sb_1 p j is_j O_j R_j D_j x_j sb_j \)
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 \( ts_{i_1} \): \( ts_{i_1} i_1 = (p_1, is_1, x_1, sb_1, D_1, O_1, R_1) \)
assume \( \text{ts-j: } \text{ts}_{sb}^j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)
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 \( \text{ts-i}_1 \) i-bound show \( ?\text{thesis} \)
  by (simp add: \( \text{ts}_{sb}' \) sb'')
next
  case False
  note \( i_1 = i \) this
  from \( \text{i-bound} \) have \( i_1 \)-bound': \( i_1 < \text{length } (\text{map owned } \text{ts}_{sb}) \)
  by auto
  from \( \text{ts-i}_1 \) False have \( \text{ts-i}_1' \): \( \text{ts}_{sb}!i_1 = (p_{i_1}, is_{i_1}, xs_{i_1}, sb_{i_1}, D_{i_1}, O_{i_1}, R_{i_1}) \)
  by (simp add: \( \text{ts}_{sb}' \) sb'')
  show \( ?\text{thesis} \)
  proof (cases \( j = i \))
    case True
    from \( \text{i-bound } \text{ts-j} \) \( \text{ts}_{sb}' \) True have \( \text{sb}_j = [] \)
    by (simp add: \( \text{ts}_{sb}' \) sb'')
    from A-unused-by-others [rule-format, OF False [symmetric]] \( \text{ts-i}_1 \) i-bound'' False have \( \text{sb}_j = [] \)
    by (auto simp add: \( \text{O}_{sb}', \text{sb}' \) owned-def)
    moreover
    from outstanding-volatile-writes-unowned-by-others [OF \( \text{i-bound}' \) i-bound i_1-i ts-i_1' ts-j']
    have \( \text{O}_{sb} \cap \text{outstanding-refs is-volatile-Write}_{sb} \) \( sb_1 = \{ \} \) by (simp add: sb)
    ultimately show \( ?\text{thesis} \) using \( \text{ts-j} \) True
    by (auto simp add: \( \text{i-bound } \text{ts}_{sb}, \text{O}_{sb}' \) sb' j)
  next
  case False
    from \( \text{j-bound} \) have \( j \)-bound': \( j < \text{length } \text{ts}_{sb} \)
    by (simp add: \( \text{ts}_{sb} \) sb')
    from \( \text{ts-j} \) False have \( \text{ts-j'} : \text{ts}_{sb}'j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)
    by (simp add: \( \text{ts}_{sb} \) sb')
    from outstanding-volatile-writes-unowned-by-others [OF \( \text{i-bound}' \) j-bound' i_1-j ts-i_1' ts-j']
    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 \( \text{ts}_{sb}' \)
proof
  fix \( n \) m
  fix \( p_n, is_n, O_n, R_n, D_n, \varnothing_n, sb_n, p_m, is_m, O_m, R_m, D_m, \varnothing_m, sb_m \)
  assume n-bound: \( n < \text{length } \text{ts}_{sb}' \)
and m-bound: m < length ts\sb' 
and neq-n-m: n \neq m 
and nth: ts\sb'n = (p_n, is_n, \varnothing_n, sb_n, D_n, O_n, R_n) 
and mth: ts\sb'm = (p_m, is_m, \varnothing_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 \text{all-acquired sb}_m) \cap 
read-only-reads (acquired True (takeWhile (Not o is-volatile-Write sb_n) sb_n) O_n) 
(dropWhile (Not o is-volatile-Write 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, \varnothing_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 
  note A-no-read-only-reads [OF n-bound' nth'] 
  ultimately 
  show ?thesis 
  using True ts\sb'-i neq-n-i nth m-bound' 
  by (auto simp add: ts\sb' O_{sb}' sb sb') 
next 
  case False 
  note neq-m-i = this 
  with m-bound mth i-bound have mth': ts\sb'm = (p_m, is_m, \varnothing_m, sb_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 i-bound nth neq-m-i n-bound' have nth': ts\sb'-m = (p_m, is_m, \varnothing_m, sb_m, D_m, O_m, R_m) 
    by (auto simp add: ts\sb'-m) 
    from read-only-reads-unowned [OF m-bound' m-bound' neq-n-m nth' mth'] False 
    note neq-m-i 
    show ?thesis 
    by (clarsimp) 
  qed 
  qed 
  next 
  show ownership-distinct 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: $i_1 < \text{length } ts_{sb}'$
assume $j$-bound: $j < \text{length } ts_{sb}'$
assume $i_1$-$j$: $i_1 \neq j$

assume $ts_i$: $ts_{sb} \upharpoonright i_1 = (p_{i_1}, is_{i_1}, xs_{i_1}, sb_1, D_1, O_1, R_1)$
assume $ts_j$: $ts_{sb} \upharpoonright j = (p_{j}, is_{j}, xs_{j}, sb_j, D_j, O_j, R_j)$

show $(O_1 \cup \text{all-acquired } sb_1) \cap (O_j \cup \text{all-acquired } sb_j) = \{\}$

proof (cases $i_1=j$)

  case True
  with $i_1$-$j$ have $i$-$j$: $i \neq j$
    by simp

  from $i$-bound $ts_i$ True have $sb_1$: $sb_1 = []$
    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'': $j < \text{length (map owned } ts_{sb})$
    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: $ts_{sb}'$)
  show $?thesis$
    proof (cases $j=i$)
      case True
      from A-unused-by-others [rule-format, OF $i$-$j$ $ts_j$ $i$-$j$ $j$-bound']
      have $A \cap (O_j \cup \text{outstanding-refs is-volatile-Write}_{sb} sb_j) = \{\}$
        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
      ultimately show $?thesis$ using $ts_i$ True
        by (auto simp add: $i$-bound $ts_{sb}'$ $O_{sb}'$ $sb' sb_1$)
    next
    case False
    note $i_1$-$i$ = 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 $ts_i'$: $ts_{sb}!i_1 = (p_{i_1}, is_{i_1}, xs_{i_1}, sb_1, D_1, O_1, R_1)$
      by (simp add: $ts_{sb}'$)
    show $?thesis$
      proof (cases $j=i$)
        case True
        from A-unused-by-others [rule-format, OF $i$-$j$ [symmetric] $ts_i$]
        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 $ts_{sb}'$ owned-def)
          moreover
579
from A-unacquired-by-others [rule-format, OF - False [symmetric]] ts-i.  False
i1-bound'

have A ∩ all-acquired sb1 = { }
by (auto simp add: Let-def ts sb′ owned-def)
moreover
from ownership-distinct [OF i1-bound' i-bound i1-1 i′ ts-i1′ ts sb-i]
have (O1 ∪ all-acquired sb1) ∩ O sb = { } by (simp add: sb)
ultimately show ?thesis

using ts-j True
by (auto simp add: i1-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′] = (p j, is j, xs j, sb j, D j, O j, R j)
by (simp add: ts sb′)
from ownership-distinct [OF i1-bound' j-bound' i1-j ts-i1′ ts-j′]
show (O1 ∪ all-acquired sb1) ∩ (O j ∪ all-acquired sb j) = { } .

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 (t→ret (m sb a) (f (θ sb (t→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′ θ sb′ sb′ sb)

qed

from valid-reads [OF i-bound ts sb-i]
have reads-consis: reads-consistent False O sb m sb sb .

have valid-reads': valid-reads m sb' ts sb'
proof (unfold-locales)
fix j p j is j O j R j D j acq j sb j
assume j-bound: j < length ts sb'
assume ts-j: ts sb′] = (p j, is 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-consis ts-j j-bound sb show ?thesis
  by (clarsimp simp add: True m sb' Write 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′] = (p j, is 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)
let \(?m' = (m_{sb}(a := f (\theta_{sb}(t \mapsto m_{sb} a))))\)
from a-unowned-others [rule-format, OF - False] j-bound
  obtain a-acq: a \notin acquired True (takeWhile (Not \circ is-volatile-Write_{sb} \ sb_j) O_j \ and
  a-unsh: a \notin all-shared (takeWhile (Not \circ is-volatile-Write_{sb} \ sb_j) by auto
with a-notin-unforwarded-non-volatile-reads-drop [OF j-bound]
  have \( \forall a \in \{\} \).\)
    \(?m' a = m_{sb} a \) by auto
from reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop [where \( W=\{\} \) simplified, OF this - - consis-m]
  have reads-consistent False O_j \( m_{sb}(a := f (\theta_{sb}(t \mapsto m_{sb} a)))) \ sb_j \)
  by (auto simp del: fun-upd-apply)
thus \( \) ?thesis by (simp add: m sb)
qed

have valid-sharing': valid-sharing \( \langle S_{sb} \oplus W \ R \ominus A \ L \rangle \) ts sb'
proof (intro-locales)
show outstanding-non-volatile-writes-unshared \( \langle S_{sb} \oplus W \ R \ominus A \ L \rangle \) ts sb'
proof (unfold-locales)
fix j p j is \( O_j \) \( R_j \) \( D_j \) \( O_j \) \( R_j \)
assume j-bound: \( j < \text{length ts}_{sb} \)
assume jth: ts sb' ! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)
show non-volatile-writes-unshared \( \langle S_{sb} \oplus W \ R \ominus A \ L \rangle \) \ 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, 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' jth']
have unshared: non-volatile-writes-unshared \( S_{sb} \ sb_j \).
have \( \forall a \in \text{dom} \langle S_{sb} \oplus W \ R \ominus A \ L \rangle - \text{dom} S_{sb} \), a \notin outstanding-refs is-non-volatile-Write_{sb} \ sb_j
  proof --
  { fix a
  assume a-in: a \in \text{dom} \langle S_{sb} \oplus W \ R \ominus A \ L \rangle - \text{dom} S_{sb}

  581
hence $a \in \mathbb{R}$ by clarsimp
assume $a \in \text{outstanding-refs is-non-volatile-Write}_{sb}$ $sb_j$
have False
proof -
  from non-volatile-owned-or-read-only-outstanding-non-volatile-writes $[\text{OF outstanding-non-volatile-refs-owned-or-read-only} \ [\text{OF j-bound } j'] ]$
    have $a \in \mathcal{O}_j \cup \text{all-acquired } sb_j$
    by auto
moreover
  with ownership-distinct $[\text{OF i-bound j-bound'} \ \text{False } ts_{sb'} \ i \ j']$ $a \in \mathbb{R}$ R-owned
show False
  by blast
qed

from non-volatile-writes-unshared-no-outstanding-non-volatile-Write $sb$
[OF unshared this]
show ?thesis .
qed

next
show sharing-consis $(\mathcal{S}_{sb} \oplus W R \ominus_{\text{A}} L) \ ts_{sb'}$
proof (unfold-locales)
  fix $j \ p_j \ is_j \ \mathcal{O}_j \ \mathcal{R}_j \ D_j \ acq_j \ xs_j \ sb_j$
  assume j-bound: $j < \text{length } ts_{sb'}$
  assume jth: $ts_{sb'} ! j = (p_j, is_j, xs_j, sb_j, D_j, \mathcal{O}_j, \mathcal{R}_j)$
show sharing-consistent $(\mathcal{S}_{sb} \oplus W R \ominus_{\text{A}} L) \ \mathcal{O}_j \ sb_j$
proof (cases i=j)
  case True
  with i-bound jth show ?thesis
  by (simp add: $ts_{sb'} \ \mathcal{sb'} \ sb$)
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': $ts_{sb} ! j = (p_j, is_j, xs_j, sb_j, D_j, \mathcal{O}_j, \mathcal{R}_j)$
  by (auto simp add: $ts_{sb'}$)
  from sharing-consis $[\text{OF j-bound'} j']$
  have consis: sharing-consistent $\mathcal{S}_{sb} \ \mathcal{O}_j \ sb_j$.

  have acq-cond: all-acquired $sb_j \cap \text{dom } \mathcal{S}_{sb} - \text{dom } (\mathcal{S}_{sb} \oplus W R \ominus_{\text{A}} L) = \{\}$
  proof -
    { 
      fix $a$
      assume a-acq: $a \in \text{all-acquired } sb_j$
      assume $a \in \text{dom } \mathcal{S}_{sb}$

582
assume \( a \in L \)

have False

proof –

from A-unacquired-by-others [rule-format, of \( j, OF - False \)] j-bound’ jth’

have \( A \cap \) all-acquired \( sb_j = \emptyset \)

by auto

with a-acq a-L L-subset

show False

by blast

qed

thus \(?thesis \)

by auto

qed

have uns-cond: all-unshared \( sb_j \cap \) dom \( (S_{sb} \oplus_W R \ominus_A L) - \) dom \( S_{sb} = \emptyset \)

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 \( sb_j \cup O_j \) 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-locales)

show \(- \bigcup (\lambda (-, -, -), (-), O). O) \cup \) set \( ts_{sb}' \) \( \subseteq \) dom \( (S_{sb} \oplus_W R \ominus_A L) \)

proof –

have s: \( \bigcup (\lambda (-, -, -, O, -), O). O) \cup \) set \( ts_{sb}' \) =

\( \bigcup (\lambda (-, -, -, -, O, -), O). O) \cup \) set \( ts_{sb} \cup A - R \)

apply (unfold ts_{sb}' O_{sb}')

apply (rule acquire-release-ownership-nth-update [OF R-owned i-bound ts_{sb-i}])
apply fact
done

note unowned-shared L-subset A-R
then
show ?thesis
  apply (simp only: s)
  apply auto
done
qed

next
show read-only-unowned \( (S_{sb} \oplus_W R \ominus_A L) \) \( t_{sb}' \)
proof
  fix j p_j is\_j O\_j R\_j D\_j acq\_j xs\_j sb\_j
  assume j-bound: \( j < \) length \( t_{sb}' \)
  assume jth: \( t_{sb}' \) ! 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 read-only-unowned [OF i-bound ts\_sb-i] R-owned A-R
  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-t')
next
  case False
  from j-bound have j-bound': \( j < \) length \( t_{sb} \)
    by (auto simp add: ts\_sb-t')
  with False jth have jth': \( t_{sb}' \) ! j = (p_j,is_j,xs_j,sb_j,D_j,O_j,R_j)
    by (auto simp add: ts\_sb-t')
  from read-only-unowned [OF j-bound' jth']
  have \( O\_j \cap read-only S_{sb} = \{ \) .
  moreover
  from A-unowned-by-others [rule-format, OF - False] j-bound' jth'
  have \( A \cap O\_j = \{ \) \)
    by (auto dest: all-shared-acquired-in )
  moreover
  from ownership-distinct [OF i-bound j-bound' False ts\_sb-i jth']
  have \( O\_sb \cap O\_j = \{ \) \)
    by auto
  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 \( (S_{sb} \oplus_W R \ominus_A L) \) \( t_{sb}' \)
proof
fix \( j \) \( p_j \) is \( j \) \( O_j \) \( R_j \) \( D_j \) acq \( x_j \) sb \( j \)
assume j-bound: \( j < \) length ts \( \sb \) \( ' \)
assume jth: ts \( \sb \) \( ' \) \( \neg \) \( j = (p_j, is_j, x_j, sb_j, D_j, O_j, R_j) \)
show no-write-to-read-only-memory (\( \mathcal{S}_{\sb} \oplus W \ R \ominus A \ L \)) sb \( j \)
proof (cases \( i = j \))
  case True
  with jth ts \( \sb \) \( i \) i-bound
  show \( \theta \)thesis
  by (auto simp add: sb sb \( \prime \) ts \( \sb \) \( \prime \))
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 \( \sb \) \( ! \) \( j = (p_j, is_j, x_j, sb_j, D_j, O_j, R_j) \)
  by (auto simp add: ts \( \sb \) \( ' \))
  from no-outstanding-write-to-read-only-memory [OF j-bound\( ' \) jth\( ' \)]
  have nw: no-write-to-read-only-memory \( \mathcal{S}_{\sb} \) sb \( j \).
  have R \( \cap \) outstanding-refs is-Write 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 \( j \) \( \cap O_{\sb} = \{ \} \)
    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 \( j \) \( \cap O_{\sb} = \{ \} \)
    by auto
    ultimately have outstanding-refs is-Write sb \( j \) \( \cap O_{\sb} = \{ \} \)
    by (auto simp add: misc-outstanding-refs-convs)
    with R-owned
    show \( \theta \)thesis by blast
  qed
  then
  have \( \forall a \in \)outstanding-refs is-Write sb \( j \).
  a \( \in \) read-only (\( \mathcal{S}_{\sb} \oplus W \ R \ominus A \ L \) \( \rightarrow \) a \( \in \) read-only \( \mathcal{S}_{\sb} \)
  by (auto simp add: in-read-only-convs)
  from no-write-to-read-only-memory-read-only-reads-eq [OF nw this]
  show \( \theta \)thesis .
  qed
  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 \( \sb \) \( ' \)
  qed

  qed
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-store-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') ∩ dom θ sb = {}
by (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 → ret (m sb a) (f (θ sb (t → m sb a))))) = {}

586
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': enough-flushs ts\sb'
by (auto simp: ts\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
    by (simp add: \sb')
  from valid-program-history-nth-update [OF i-bound causal' this]
  show ?thesis
    by (simp add: ts\sb' \sb' \sb')
  qed

from is-sim have 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 \varnothing=\varnothing sb and m=m, OF cond']
have (RMW a t (D, f) cond ret A L R W # is\sb', \varnothing sb, (), m, D, \O sb, R, S) →
  (is\sb',\varnothing sb(t ↦ ret (m a) (f (\varnothing sb(t→m a)))), (),
   m(a := f (\varnothing sb(t ↦ m a)))), False, \O sb U A − R, Map.empty, S ⊕ W R ⊕ A L).

from direct-computation.concurrent-step.Memop [OF i-bound' ts-i this]
have (ts, m, S) ⇒d (ts[i := (p\sb, is\sb',\varnothing sb(t ↦ ret (m a) (f (\varnothing sb(t→m a)))), ()], False, 
O sb U A − R, Map.empty],
   m(a := f (\varnothing sb(t ↦ m a)))), S ⊕ W R ⊕ A L).

moreover 

have tmps-commute: \varnothing sb(t ↦ ret (m\sb a) (f (\varnothing sb(t→m\sb a)))) =
  (\varnothing sb |' (dom \varnothing sb − {t}))(t ↦ ret (m\sb a) (f (\varnothing sb(t→m\sb a))))
apply (rule ext)
apply (auto simp add: restrict-map-def domIf)
done

from a-unflushed ts\sb-i sb
have a-unflushed':
  \forall j < \text{length ts}_{sb}.
   (let (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = ts\sb!j
    in a \not\in \text{outstanding-refs is-non-volatile-Write}_{sb} (\text{takeWhile} (\text{Not} \circ
    \text{is-volatile-Write}_{sb}) s_{bj}))
by auto

have all-shared-L: \( \forall i \ p \in O \ R \ D \ \text{acq} \ \emptyset \ \text{sb}. \ i < \text{length} \ ts_{sb} \ \rightarrow \ \\)
\( ts_{sb} \! \! (p, \ \text{is}, \ \emptyset, \ \text{sb}, \ D, \ O, R) \ \rightarrow \ \text{all-shared} (\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}) \cap L = \{\} \)
proof -
{ 
fix \ j \ p_j \ is_j \ O_j \ R_j \ D_j \ \emptyset_j \ \text{sb}_j \ x
assume \ j-bound: \ j < \text{length} \ ts_{sb}
assume \ jth: \ ts_{sb}^j = (p_j, \text{is}_j, \emptyset_j, \text{sb}_j, \text{D}_j, \text{O}_j, \text{R}_j)
assume \ x-shared: \ x \in \text{all-shared} (\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}_j)
assume \ x-L: \ x \in L
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 -
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 \text{O}_j.
moreover have all-shared (takeWhile (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}_j) \subseteq \text{all-shared} \text{sb}_j
using all-shared-append [of (takeWhile (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}_j)
\ (\text{dropWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{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 \text{O}_j = \{\}
by (auto dest: all-shared-acquired-in)
ultimately
show False
using L-subset x-L, x-shared
by blast
qed
qed
}
thus \ ?thesis by blast
qed

have all-shared-A: \( \forall i \ p \in O \ R \ D \ \emptyset \ \text{sb}. \ i < \text{length} \ ts_{sb} \ \rightarrow \ \\)
\( ts_{sb} \! \! (p, \ \text{is}, \ \emptyset, \ \text{sb}, \ D, \ O, R) \ \rightarrow \ \text{all-shared} (\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}) \cap A = \{\} \)
proof -
{  
  fix \( j \) \( p_j \) \( is_j \) \( OR_j \) \( D_j \) \( \theta_j \) \( sb_j \) \( x \)  
  assume j-bound: \( j < \) length \( ts_{sb} \)  
  assume jth: \( ts_{sb}[j] = (p_j, is_j, \theta_j, sb_j, OR_j, D_j) \)  
  assume x-shared: \( x \in \) all-shared (takeWhile (Not \circ is-volatile-Write_{sb}) \( sb_j \))  
  assume x-A: \( x \in A \)  
  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  
      from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]  
      have all-shared \( sb_j \subseteq \) all-acquired \( sb_j \cup OR_j \).  
      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  
      from A-unacquired-by-others [rule-format, OF - False] jth j-bound  
      have \( A \cap \) all-acquired \( sb_j \) = {} by auto  
      moreover  
      from A-unowned-by-others [rule-format, OF - False] jth j-bound  
      have \( A \cap OR_j \) = {}  
      by (auto dest: all-shared-acquired-in)  
    ultimately  
    show False  
    using x-A x-shared  
    by blast  
    qed  
    qed  
}  
thus ?thesis by blast  
  qed  
  hence all-shared-L: \( \forall i \) \( p \) is \( OR \) \( DR \) \( \theta \) \( sb \). \( i < \) length \( ts_{sb} \) \( \rightarrow \)  
  \( ts_{sb} \). \( i = (p, is, \theta, sb, DR, OR, \theta) \) \( \rightarrow \)  
  all-shared (takeWhile (Not \circ is-volatile-Write_{sb}) \( sb \)) \( \cap \) L = {}  
  using L-subset by blast  
  have all-unshared-R: \( \forall i \) \( p \) is \( OR \) \( DR \) \( \theta \) \( sb \). \( i < \) length \( ts_{sb} \) \( \rightarrow \)  
  \( ts_{sb} \). \( i = (p, is, \theta, sb, DR, OR, \theta) \) \( \rightarrow \)  
  all-unshared (takeWhile (Not \circ is-volatile-Write_{sb}) \( sb \)) \( \cap \) R = {}  
  proof  
  {  

fix \ j \ p_j \ is_j \ O_j \ R_j \ 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, D_j, O_j, R_j)
assume x-unshared: \ x \in \text{all-unshared} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_j)
assume x-R: \ x \in R
have False
proof (cases i=j)
  case True with x-unshared ts_{sb}-i jth show False by (simp add: sb)
next
  case False
  show False
  proof
    from unshared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]
    have all-unshared sb_j \subseteq \text{all-acquired} sb_j \cup O_j.
    moreover have all-unshared (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb_j) \subseteq all-unshared sb_j
    using all-unshared-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
    note ownership-distinct [OF i-bound j-bound False ts_{sb}-i jth]
    ultimately
    show False
    using R-owned x-R x-unshared
    by blast
    qed
    qed
  qed

thus \ \text{thesis} by blast
qed

have all-acquired-R: \ \forall \ p \ in O \ R \ D \ \emptyset \ sb. \ i < \ \text{length} \ ts_{sb} \rightarrow
  ts_{sb} ! i = (p, is, \emptyset, sb, D, O, R) \rightarrow
  \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ sb) \cap R = \{}
proof
  { fix \ j \ p_j \ is_j \ O_j \ R_j \ 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, D_j, O_j, R_j)
    assume x-acq: \ x \in \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ 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
  qed
proof

from x-acq have x ∈ all-acquired sby using all-acquired-append [of takeWhile (Not ◦ is-volatile-Write sby) sby dropWhile (Not ◦ is-volatile-Write sby) sby]
by auto
moreover
note ownership-distinct [OF i-bound j-bound False tsb-i jth]
ultimately
show False
using R-owned x-R
by blast
qed
qed

thus ?thesis by blast
qed

have all-shared-R: ∀ i p is ORD θ sb. i < length tsb ! i = (p, is, 0, sb, D, O, R) →
tsb ! i = (p, is, 0, sb, D, O, R) →
all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∩ R = {}

proof –

{ fix j p̄j isj Oj Rj D̄j 0̄j sby j x
assumption j-bound: j < length tsb
assume jth: tsb ! j = (p̄j, isj, 0̄j, sbj, D̄j, Oj, Rj)
assume x-shared: x ∈ all-shared (takeWhile (Not ◦ is-volatile-Write sby) sby)
assume x-R: x ∈ R
have False
proof (cases i=j)
case True with x-shared tsb-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]]

moreover have all-shared (takeWhile (Not ◦ is-volatile-Write sby) sby) ⊆ all-shared sby
using all-shared-append [of (takeWhile (Not ◦ is-volatile-Write sby) sby)
(dropWhile (Not ◦ is-volatile-Write sby) sby)]
by auto
moreover
note ownership-distinct [OF i-bound j-bound False tsb-i jth]
ultimately
show False
using R-owned x-R x-shared
by blast
qed
thus \textit{thesis} by blast

\textbf{qed}

\textbf{from} share-all-until-volatile-write-commute \[ \text{OF (ownership-distinct ts}_{sb}\text{)} \]
(share-consis \( S_{sb} \) \( ts_{sb}\))
all-shared-L all-shared-A all-acquired-R all-unshared-R all-shared-R
\textbf{have} share-commute: share-all-until-volatile-write \( ts_{sb} \) \( S_{sb} \oplus_W R \ominus_A L = \)
share-all-until-volatile-write \( ts_{sb} \) \((S_{sb} \oplus_W R \ominus_A L)\).

\{
\begin{align*}
\text{fix} & \ j \ p_j \ is_j \ O_j \ R_j \ D_j \ \vartheta_j \ sb_j \ x \\
\text{assume} & \ jth: \ ts_{sb}|j = (p_j, is_j, \vartheta_j, sb_j, D_j, O_j, R_j) \\
\text{assume} & \ j-bound: \ j < \text{length} \ ts_{sb} \\
\text{assume} & \ neq: \ i \neq j \\
\textbf{have} & \ \text{release (takeWhile (Not \circ is-volatile-Write}_{sb}\text{) sb}_{j))} \\
& \\quad \text{dom} \ S_{sb} \cup R - L) \ R_j \\
& \quad = \text{release (takeWhile (Not \circ is-volatile-Write}_{sb}\text{) sb}_{j))} \\
& \quad \text{dom} \ S_{sb} \ R_j \\
\textbf{proof} – \\
\begin{aligned}
& \\quad \text{fix} \ a \\
& \\quad \text{assume} \ a-in: \ a \in \text{all-shared (takeWhile (Not \circ is-volatile-Write}_{sb}\text{) sb}_{j))} \\
& \\quad \text{have} \ (a \in \text{dom} \ S_{sb} \cup R - L)) = (a \in \text{dom} \ S_{sb}) \\
& \\quad \textbf{proof} – \\
& \quad \text{from} \ A-unowned-by-others \ [\text{rule-format, OF j-bound neq }] \ jth \\
& \quad \text{A-unacquired-by-others} \ [\text{rule-format, OF - neq]} \ j-bound \\
& \quad \textbf{have} \ A-dist: \ A \cap (O_j \cup \text{all-acquired sb}_{j}) = \{\} \\
& \quad \textbf{by} (\text{auto dest: all-shared-acquired-in}) \\
& \textbf{from} \ \text{all-shared-acquired-or-owned} \ [\text{OF sharing-consis [OF j-bound jth]}] \ a-in \\
& \text{all-shared-append} \ [\text{of (takeWhile (Not \circ is-volatile-Write}_{sb}\text{) sb}_{j})] \\
& \textbf{have} \ a-in: \ a \in O_j \cup \text{all-acquired sb}_{j} \\
& \textbf{by} \ \text{auto} \\
& \textbf{with} \ \text{ownership-distinct} \ [\text{OF i-bound j-bound neq ts}_{sb}-i jth] \\
& \textbf{have} \ a \notin (O_{sb} \cup \text{all-acquired sb}) \ \textbf{by} \ \text{auto} \\
& \textbf{with} \ A-dist R-owned A-R A-shared-owned L-subset a-in \\
& \textbf{obtain} \ a \notin R \ \textbf{and} \ a \notin L \\
& \textbf{by} \ \text{fastforce} \\
& \textbf{then show} \ \textit{thesis} \ \textbf{by} \ \text{auto} \\
\textbf{qed} \\
\end{aligned}
\end{align*}
\}

\textbf{apply} – 

\textbf{qed}
apply (rule release-all-shared-exchange)
apply auto
done

qed

note release-commute = this
have \[(\text{ts}_{sb}, \text{ms}_{sb}(a := f (\vartheta_{sb}(t \mapsto m_{sb} a))), \mathcal{S}_{sb}) \sim (\text{ts}[i := (p_{sb}, is_{sb}', \theta_{sb}(t \mapsto \text{ret}(m a))(f (\vartheta_{sb}(t \mapsto m a))))],(), \text{False}, \mathcal{O}_{sb} \cup \mathcal{A} - \mathcal{R}, \text{Map.empty}], m(a := f (\vartheta_{sb}(t \mapsto m a))), \mathcal{S} \oplus \mathcal{W} \ominus \mathcal{A} \ominus \mathcal{L})\]
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}' \theta_{sb}' \mathcal{D}_{sb}' ex-not m-a split: if-split-asm)
apply (rule conjI)
apply clarsimp
apply (rule tmps-commute)
apply clarsimp
apply (rule (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}' \mathcal{S}_{sb}'
by (auto simp del: fun-upd-apply)

next
case (SBHGhost A L R W)
then obtain
is_{sb}: is_{sb} = Ghost A L R W# is_{sb}' and
\mathcal{O}_{sb}': \mathcal{O}_{sb} = \mathcal{O}_{sb} and
\mathcal{R}_{sb}': \mathcal{R}_{sb} = \mathcal{R}_{sb} and
\vartheta_{sb}': \vartheta_{sb} = \vartheta_{sb} and
\mathcal{D}_{sb}': \mathcal{D}_{sb} = \mathcal{D}_{sb} and
sb': sb'=sb@[\text{Ghost}_{sb} A L R W] and
m_{sb}': m_{sb} = m_{sb} and
\[ S_{\text{sb}} : S_{\text{sb}}' = S_{\text{sb}} \]

by auto

from safe-memop-flush-sb [simplified is_{\text{sb}}] obtain

L-subset: \( L \subseteq A \) and

A-shared-owned: \( A \subseteq \text{dom (share \ ?drop-sb } S) \) \( \cup \) acquired True \( \text{sb } O_{\text{sb}} \) and

R-acq: \( R \subseteq \text{acquired True } \text{sb } O_{\text{sb}} \) and

A-R: \( A \cap R = \{\} \) and

A-unowned-by-others-ts:

\[ \forall j < \text{length } (\text{map owned ts}), i \neq j \rightarrow (A \cap (\text{owned } (\text{ts}[j]) \cup \text{dom } (\text{released } (\text{ts}[j]))) = \{\}) \]

by cases auto

from A-unowned-by-others-ts ts-sim leq

have A-unowned-by-others:

\[ \forall j < \text{length } \text{ts}_{\text{sb}}, i \neq j \rightarrow \text{(let } (\text{O}_j, \text{sb}_j) = \text{map } \text{O}-sb \text{ ts}_{\text{sb}}!j \text{ in } A \cap \text{outstanding-refs } \text{is-volatile-Write}_{\text{sb}} \text{ sb}_j = \{\}) \]

proof -

\{ fix j \text{ O}_j \text{ sb}_j assume j-bound: j < \text{length } (\text{map owned } ts_{\text{sb}})\}

assume neq-i-j: i \neq j

assume ts_{sb}: (\text{map } \text{O}-sb \text{ ts}_{\text{sb}})!j = (\text{O}_j, \text{sb}_j)

assume conflict: A \cap \text{outstanding-refs } \text{is-volatile-Write}_{\text{sb}} \text{ 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_{\text{sb}}

by auto

from j-bound have j-bound: j < \text{length } ts

by simp

from conflict obtain a' where

\( a' \text{-in: } a' \in A \) and

\( a' \text{-in-j: } a' \in \text{outstanding-refs } \text{is-volatile-Write}_{\text{sb}} \text{ sb}_j \)

by auto

let ?take-sb: = (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}} \text{ sb}_j))

594
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 i sb j D j R j is j where
tsb j: ts sb j ! j = (p j, is sb j, sb j, D 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 sb j ≠ {}) and

is j: instrs suspends j @ is sb j = is sb j @ prog-instrs suspends 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
qed

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 =is sb and m=flush ?drop-sb m]

have (Ghost A L R W # is sb',

\begin{align*}
&\hat{\vartheta}_{sb}, (), \text{flush } ?\text{drop-sb } m, D_{sb}, \\
&\quad \text{acquired True } sb \ O_{sb}, \text{release } sb \ (\text{dom } S_{sb}) \ R_{sb}, \text{share } ?\text{drop-sb } S \\
&\quad \rightarrow (is sb', \hat{\vartheta}_{sb}, () , \text{flush } ?\text{drop-sb } m, D_{sb}, \\
&\quad \text{acquired True } sb \ O_{sb} \cup A \rightarrow R, \\
&\quad \text{augment-rels } (\text{dom } (\text{share } ?\text{drop-sb } S ) ) \ R \ (\text{release } sb \ (\text{dom } S_{sb}) \ R_{sb}), \\
&\quad \text{share } ?\text{drop-sb } S \oplus_W R \oplus_A L).
\end{align*}

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, flush ?drop-sb m, share ?drop-sb S) \Rightarrow_d

\begin{align*}
&\begin{cases}
&\text{dom } (\text{share } ?\text{drop-sb } S ) \ R \ (\text{release } sb \ (\text{dom } S_{sb}) \ R_{sb})), \\
&\quad \text{flush } ?\text{drop-sb } m, \text{share } ?\text{drop-sb } S \oplus_W R \oplus_A L) \\
&\text{is } - \Rightarrow_d (?ts-A, ?m-A, ?share-A)
\end{cases}
\end{align*}

by (simp add: is sb)
from \(i\)-bound' have \(i\)-bound'': \(i < \text{length } ?ts-A\)
  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 True a' sop v A' L' R' W' # zs)), is_j, \(\theta_{sbj} l' (\text{dom } \theta_{sbj} - \text{read-tmps} (ys @ Write True a' sop v A' L' R' W' # zs)), (, D_j, acquired True ?take-sb\(j\) \(O\), 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'''': \(j < \text{length } ?ts-A\)
  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 \(\text{False } O_j m_{sb} sb_j\).

  from reads-consistent-flush-all-until-volatile-write [OF \(\text{valid-ownership-and-sharing } S_{sb} ts_{sb-j'} j\)-bound''
\(ts_{sb}-j\) reads-consis]
have reads-consis-m: reads-consistent \(\text{True } (\text{acquired True } ?\)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 \(\cap\) 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 \(\text{True } (\text{acquired True } ?\)take-sb\(j\) \(O\) j) \(\text{?m-A suspends\(j\)}\),

596

from valid-history [OF j-bound'' ts_{sb-j}]
have h-consis:
  history-consistent \( \partial_{sb_j} \) (hd-prog p_j (?take-sb_j@suspends_j)) (?take-sb_j@suspends_j)
  apply (simp only: split-suspends_j [symmetric] suspends_j)
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 \( \partial_{sb_j} \) (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-ref s is-volatile-Read \( j \) \( \ys \) = \{ \}
  by (auto simp add: outstanding-ref s-append suspends \( j \) [symmetric]
  split-suspends \( j \))

from flush-store-buffer-append [
  OF \( j \)-bound''' \( \ys \) = \{ \}
  causal-program-history \( \ys \) \{ \}
  acquired True \( \ys \)
  flush \( \ys \)
  share \( \ys \)
  interpret causal \( \ys \) \{ \}
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 \( \ys \) \{ \}
  by simp
from causal-program-history-suffix [OF this]
have causal-program-history \( \ys \) \{ \}
interpret causal \( \ys \) \{ \}
by (rule cph')

from causal \( \ys \) causal-program-history [of \( \ys \), simplified, OF refl] \( \ys \)
obtain is \( j \)'''
  where \( is \)\( j \)'' = Write True \( a'\) sop A' \( L'\) R' \( W'\) \# \( \ys \)''
  is \( j \)''': instrs \( \ys \) \# \( \ys \)\( j \) sbj = is \( a\)'' 
  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’,

θ sbj |’ (dom θ sbj − read-tmps (Write sb True a’ sop v A’ L’ R’ W’# zs)),()

D_j ∨ outstanding-refs is-volatile-Write sb ys ≠ {},

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:
∀ i < length ?ts-ys. j≠i → (let (O_i) = map owned ?ts-ys!i in a’ ∉ 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:
∀ j<length (map O-sb ts sb). i ≠ j →

(let (O_j, sb_j) = map O-sb ts sb! j
in A ∩ 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≠j
assume ts sb-j: (map O-sb ts sb)!j = (O_j, sb_j)
assume conflict: A ∩ all-acquired 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’ ∈ all-acquired sb_j
by auto

599
let \( \text{take-sb}_j = (\text{takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \text{sb}}_j) \)

let \( \text{drop-sb}_j = (\text{dropWhile (Not \circ \text{is-volatile-Write}_{sb}) \text{sb}}_j) \)

from \( \text{ts-sim} \) [rule-format, OF \( j \)-bound”]

obtain \( p_j \) suspends\( j \) \( is_{sbj} \) \( \emptyset_{sbj} \) \( D_{sbj} \) \( R_j \) \( is_j \) where

\( ts_{sbj}: ts_{sb} \! j \! j = (p_j, is_{sbj}, \emptyset_{sbj}, D_{sbj}, O_j, R_j) \) and

\( \text{suspends}_j: \text{suspends}_j = ?\text{drop-sb}_j \) and

\( D_j: D_{sbj} = (D_j \lor \text{outstanding-refs is-volatile-Write}_{sb} \text{sb}_j \neq \{\}) \) and

\( is_j: \text{instrs suspends}_j @ is_{sbj} = is_j @ \text{prog-instrs suspends}_j \) and

\( ts_j: ts_{lj} = (\text{hd-prog } p_j \text{ suspends}_j, is_j, (\text{dom } \emptyset_{sbj} - \text{read-tmps suspends}_j), (), D_j, \text{acquired True } ?\text{take-sb}_j O_j, \text{release } ?\text{take-sb}_j (\text{dom } S_{sb}) R_j) \)

apply (cases \( ts_{sbj} \! j \! j \))

apply (force simp add: Let-def)
done

from \( a' \)-in-j all-acquired-append [of \( ?\text{take-sb}_j ?\text{drop-sb}_j \)]

have \( a' \in \text{all-acquired } ?\text{take-sb}_j \lor a' \in \text{all-acquired } \text{suspends}_j \)
by (auto simp add: \( \text{suspends}_j \))

thus False
proof
assume \( a' \in \text{all-acquired } ?\text{take-sb}_j \)
with A-unowned-by-others [rule-format, OF - neq-i-j \( ts_{sbj} \! j \! j \)-bound \( a' \)-in]
show False
by (auto dest: all-acquired-unshared-acquired)

next
assume conflict-drop: \( a' \in \text{all-acquired } \text{suspends}_j \)
from split-all-acquired-in [OF conflict-drop]

show False
proof
assume \( \exists\text{sop a'' v y z s a L R W.} \)

\( \text{suspends}_j = ys @ \text{Write}_{sb} \) True \( a'' \) \( \text{sop} \) \( v \) \( A \) \( L \) \( R \) \( W \) \( # \) \( zs \) \( a' \in A \)

then
obtain \( a'' \) \( \text{sop' v' y z s a' L' R' W'} \) where

\( \text{split-suspends}_j: \text{suspends}_j = ys @ \text{Write}_{sb} \) True \( a'' \) \( \text{sop' v' A' L' R' W'} \) \( # \) \( zs \)

(is suspends\( j = ?\text{suspends}) \) and

\( a' \) \( A' \), \( a' \in A' \)
by auto

from direct-memop-step.Ghost [where \( \emptyset = \emptyset_{sb} \) and \( m = \text{flush } ?\text{drop-sb} \ m \)]

have (Ghost \( A \) \( L \) \( R \) \( W \) \# \( is_{sb} \),

\( \emptyset_{sb}, (), \) \( \text{flush } ?\text{drop-sb} \ m, D_{sb}, \) acquired True \( sb \) \( O_{sb} \), release \( sb \) \( (\text{dom } S_{sb}) R_{sb}, \text{share } ?\text{drop-sb} S \) \rightarrow

\( (is_{sb}, \emptyset_{sb}, (), \) \( \text{flush } ?\text{drop-sb} \ m, D_{sb}, \) acquired True \( sb \) \( O_{sb} \cup A - R, \)

augment-rels \( (\text{dom } (\text{share } ?\text{drop-sb} S)) R \) \( \text{release } sb \) \( (\text{dom } S_{sb}) R_{sb}, \)

\text{share } ?\text{drop-sb} S \oplus W \) \( R \ominus A \) \( L \).

600
from direct-computation.concurrent-step.Menop \[OF i-bound-ts'\] ts'-i [simplified is_{sb}] this [simplified is_{sb}]

**have** store-step: (?ts', flush ?drop-sb m, share ?drop-sb S) \(\Rightarrow_d\)

\[(?ts'i := (p_{sb}, i_{sb}', \emptyset_{sb}, ()), D_{sb}, \]

\[\text{acquired True sb } O_{sb} \cup A - R, \]

\[\text{augment-rels (dom (share ?drop-sb S)) R (release sb (dom } S_{sb}) R_{sb}))],\]

\[\text{flush ?drop-sb m, share ?drop-sb } S \oplus W R \ominus A L)\]

\[(is - \Rightarrow_d (?ts-A, ?m-A, ?share-A) )\]

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} sbj,

**then** have cph: causal-program-history is_{sbj} ?suspends

apply –

apply (rule causal-program-history-suffix \[where \text{sb=?take-sb}_{j}] )

apply \(\text{simp only: split-suspends}_{j} \ [\text{symmetric} \text{suspends}_{j}]\)

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),

\[i_{j}, \]

\[\emptyset_{sbj} |^i \text{ (dom } \emptyset_{sbj} - \text{ read-tmps (ys @ Write}_{sb} \text{ True a'' sop' v' A' L' R' W' # zs))}, ()],\]

\[D_{j}, \]

\[\text{acquired True } ?\text{take-sb}_{j} O_{j}, \text{ release } ?\text{take-sb}_{j} (\text{dom } S_{sb}) R_{j})\]

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 \text{sb=?take-sb}_{j}] )

apply \(\text{simp only: split-suspends}_{j} \ [\text{symmetric} \text{suspends}_{j}]\)

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} \cdot ts_{sb} \cdot j-bound''

have reads-consis-m: reads-consistent True (acquired True ?take-sb \midd j \midd O)
by (simp add: m suspends_j)

from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound \midd ts_{sb-i} \midd ts_{sb-j}]
have outstanding-refs is-Write_{sb} \midd drop-sb \cap outstanding-refs is-non-volatile-Read_{sb}
suspends_j = {}
by (simp add: suspends)
have m-A suspends_j.
hence reads-consis-m-A-ys: reads-consistent True (acquired True ?take-sb \midd O)
\midd ?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 \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 \midd 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

602
from read-tmps-distinct \[\text{OF j-bound}'' ts_{sb\cdot j}\]
have distinct-read-tmps \(?\text{take-sb}_j \cap \text{suspends}_j\) 
by simp add: split-suspends \[j\text{ symmetric} \] suspends \[j\]
then obtain
read-tmps-take-drop: read-tmps \(?\text{take-sb}_j \cap \text{suspends}_j\) = \{\} and
distinct-read-tmps-drop: distinct-read-tmps suspends \[j\]
apply simp only: split-suspends \[j\text{ symmetric} \] suspends \[j\]
apply simp only: distinct-read-tmps-append
done

from history-consistent-appendD \[\text{OF valid-sops-take read-tmps-take-drop h-consis}\]
last-prog-hd-prog
have hist-consis': history-consistent \(\theta_{sbj} (\text{hd-prog p}_j \text{suspends}_j)\) suspends \[j\]
by simp add: split-suspends \[j\text{ symmetric} \] suspends \[j\]
from reads-consistent-drop-volatile-writes-no-volatile-reads
have no-vol-read: outstanding-refs is-volatile-Read \[sbys\] = \{\}
by (auto simp add: outstanding-refs-append suspends \[j\text{ symmetric} \] split-suspends \[j\])
from flush-store-buffer-append \[\text{OF j-bound}''' \]
is \[S_{\text{A}\cdot j}\] suspends \[j\]
ct \[S_{\text{A}\cdot j}\text{ simplified split-suspends} \[j\]
ct \[S_{\text{A}\cdot j}\text{ simplified split-suspends} \[j\]
read-consis-m-A-ys
hist-consis' \[\text{symmetric split-suspends} \[j\]
valid-sops-drop distinct-read-tmps-drop
[simplified split-suspends]
no-volatile-Read_{sb\cdot y} volatile-reads-consistent \[\text{OF no-vol-read}, \text{where}\]
\(S = ?\text{share-A}\)
obtain is\[j'\] \(R_j'\) where
is \[j'\], instrs \(\text{Write}_{sb} \text{ True a'' sop' v' A' L' R' W' # zs}\) @ is\[sbj\] =
is \[j'\] @ prog-instrs \(\text{Write}_{sb} \text{ True a'' sop' v' A' L' R' W' # zs}\)
and
steps-ys: (?ts-A \[?m-A, ?share-A\] \rightarrow d^* )
(?ts-A[\[j\): (last-prog \(\text{hd-prog p}_j \text{(Write}_{sb} \text{ True a'' sop' v' A' L' R' W' # zs)}\)) ys,
is \[j'\],
\(\partial_{sbj} \cap \text{read-tmps (Write}_{sb} \text{ True a'' sop' v' A' L' R' W' # zs)}\) , \() ,
\(D_j \cup \text{outstanding-refs is-volatile-Write}_{sb} \text{ ys} \neq \{\}, \text{acquired True ys}
\(\text{acquired True } ?\text{take-sb}_j O_{j}), R_j'\) \],
flush ys ?m-A,share ys ?share-A)
(is \(\cdot\cdot\cdot \rightarrow d^* (?ts-ys,?m-ys,?shared-ys)\))
by (auto)

note conflict-computation = rtranclp-trans \[\text{OF rtranclp-r-rtranclp} \[\text{OF steps-flush-sb, OF store-step} \] steps-ys]
from cph
have causal-program-history is\[sbj\] \((ys @ \text{Write}_{sb} \text{ True a'' sop' v' A' L' R' W'}) @ zs\)
by simp
from causal-program-history-suffix \[\text{OF this}\]
have cph': causal-program-history is\[sbj\] zs.
interpret causal, causal-program-history is tz by (rule cph')

from causal, causal-program-history [of [], simplified, OF refl] is’

obtain is’''

where is’; is’ = Write True a'' sop' A' L' R' W'[:is’'' and
is’''': instrs zs @ is’ = is’'' @ 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 (Write sb True a'' sop’ A’ L’ R’ W’ zs)) ys, is’’;

\theta_{sbj} | (dom \theta_{sbj} - read-tmps (Write sb True a'' sop' A' L' R' W' zs)).(),

D_j \lor outstanding-refs is-volatile-Write sb ys \neq \{\},

acquired True ys (acquired True ?take-sb 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’]

have A'-unowned:

\forall i < length ?ts-ys. j\neq i \rightarrow (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 \theta' = \theta sb and m=flush ?drop-sb m]

have (Ghost A L R W # is sb',

\theta 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', \theta sb, (), flush ?drop-sb m, D sb,

acquired True sb O sb \cup A \rightarrow R,

augment-rels (dom (share ?drop-sb S)) R (release sb (dom S sb) R sb),

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) ⇒_d
(?(ts′!i := (p_{sb}, is_{sb′}, θ_{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 ⊕ W R ⊖ 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_{sb} j sb_j.
then have cph: 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_{sb})
apply (simp add: split-suspends_{sb})
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), is_{j},
θ_{sb} | (dom θ_{sb} – read-tmps (ys @ Ghost_{sb} A’ L’ R’ W’# zs)), (),D_{j},
aquired True ?take-sb_{j} O_{j},release ?take-sb_{j} (dom S_{sb} R_{j})
by (simp add: split-suspends_{sb})

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])
apply (simp only: split-suspends_{sb} [symmetric] suspends_{sb})
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} j-bound″
ts_{sb,j} reads-consis]
have reads-consis-m: reads-consistent True (acquired True ?take-sb_{j} O_{j}) m suspends_{sb}
by (simp add: m suspends_{sb})
from outstanding-non-write-non-vol-reads-drop-disj [OF i-bound j-bound" neq-i-j ts_{sb} ti 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 ?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 \theta_{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_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

done

from read-tmps-distinct [OF j-bound" ts_{sb} j]
have distinct-read-tmps (?take-sb_j @suspends_j)
and (simp add: split-suspends_j [symmetric] suspends_j)
then obtain
read-tmps-take: read-tmps ?take-sb_j \cap read-tmps suspends_j = {} and
apply (simp only: split-suspendsj [symmetric] suspendsj)
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 θsbj (hd-prog pj suspendsj) suspendsj
by (simp add: split-suspendsj [symmetric] suspendsj)
from reads-consistent-drop-volatile-writes-no-volatile-reads
[OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Readsbys = {}
by (auto simp add: split-suspendsj [symmetric]
  split-suspendsj)
from flush-store-buffer-append [OF j-bound'''' isj [simplified split-suspendsj] cph [simplified suspendsj]
  hist-consis' [simplified split-suspendsj] valid-sops-drop distinct-read-tmps-drop
  [simplified split-suspendsj]
  no-volatile-Readsb-volatile-reads-consistent [OF no-vol-read], where
  S=?share-A]
  obtain isj'' Rj' where
  isj'': instrs (Ghostsb A' L' R' W'# zs) @ issbj =
  isj' @ proc-instrs (Ghostsb A' L' R' W'# zs) and
  steps-ys: (?ts-A, ?m-A, ?share-A) ⇒d''
  (?ts-A[j]:= (last-prog (hd-prog pj (Ghost sb A' L' R' W'# zs))) ys,
  isj',
  θsbj |' (dom θsbj - read-tmps (Ghost sb A' L' R' W'# zs)),(),
  Dj \ union outstanding-refs is-volatile-Write sbys \neq {}, acquired True ys (acquired
  True ?take-sbq Oj'),Rj') ],
  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 issbj ((ys @ [Ghostsb A' L' R' W']) @ zs)
by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history issbj zs.
interpret causalj: causal-program-history issbj zs by (rule cph')

from causalj.causal-program-history [of [], simplified, OF refl] isj'
obtain isj''
where isj': isj' = Ghost A' L' R' W'#isj'' and
isj'': instrs zs @ issbj = isj'' @ proc-instrs zs
by clarsimp

607
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 (Ghost sb A'L'R'W'# zs)) ys, isj',
    dom sbj |' (dom 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),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'-unowned:
  \forall i < length ?ts-ys. j \neq i \rightarrow (let (O_j) = map owned ?ts-ys!i in A' \cap O_i = \{\})
apply cases
apply (fastforce simp add: Let-def is sb)
done
proof
  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

have A-no-read-only-reads-by-others:
\forall j < length (map O-sb ts sb). i \neq j \rightarrow
  (let (O_j, sb_j) = map O-sb ts sb! j
  in A \cap 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) = \{\})
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)
  let ?take sb j = (takeWhile (Not o is-volatile-Write sb) sb_j)
  let ?drop sb j = (dropWhile (Not o is-volatile-Write sb) sb_j)

  assume conflict: A \cap read-only-reads (acquired True ?take sb j O_j) ?drop sb j \neq \{\}
  have False
  proof

608
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-sb_j O_j) ?drop-sb_j
  by auto

from ts-sim [rule-format, OF j-bound'''] ts_sbj-j j-bound''
obtain p_j suspendedj is_sbj D_sbj D_j \varnothing_sbj i_sj where
  ts_sbj-j: ts_sbj ! j = (p_j, is_sbj, \varnothing_sbj, sb_j, D_sbj, O_j, R_j) and
  suspendedsj: suspendedsj = ?drop-sb_j and
  i_sj: instrs suspendedsj @ is_sbj = i_sj @ prog-instrs suspendedsj and
  D_j: D_sbj = (D_j \lor outstanding-refs is-volatile-Write sb_j \neq \{\}) and
  ts_j: ts_j! = (hd-prog p_j suspendedsj, i_sj,
    \varnothing_sbj |! (dom \varnothing_sbj - read-tmps suspendedsj),(), D_j, acquired True ?take-sb_j O_j, release
  ?take-sb_j (dom S_sbj R_j))
  apply (cases ts_sbj)
  apply (force simp add: Let-def)
done

from split-in-read-only-reads [OF a'-in-j [simplified suspendedsj [symmetric]]]
obtain t v ys zs where
  split-suspendedsj: suspendedsj = ys @ Read sb_False a' t v# zs (is suspendedsj = ?suspends)
and
  a'-unacq: a' \notin acquired True ys (acquired True ?take-sb_j O_j)
  by blast

from direct-memop-step.Ghost [where \varnothing=\varnothing_sbj and m=flush ?drop-sb m]
have (Ghost A L R W# is_sbj',
  \varnothing_sbj, (), flush ?drop-sb m, D_sbj,
  acquired True sb O_ab, release sb (dom S_sbj) R_ab, share ?drop-sb S) →
  (is_sbj', \varnothing_sbj, (), flush ?drop-sb m, D_sbj,
  acquired True sb O_ab \cup A - R,
  augment-rels (dom (share ?drop-sb S)) R (release sb (dom S_sbj) R_ab),
  share ?drop-sb S @W R @A L).

from direct-computation.concurrent-step.Memop [OF
i-bound-ts'[simplified is_sbj] ts'-i [simplified is_sbj] this [simplified is_sbj]]
have store-step: (?ts', flush ?drop-sb m, share ?drop-sb S) ⇒_d
  (?ts'[i] := (p_ab, is_sbj, \varnothing_sbj, (), D_sbj, acquired True sb O_ab \cup A - R, augment-rels
  (dom (share ?drop-sb S)) R (release sb (dom S_sbj) R_ab))],
flush ?drop-sb m,share ?drop-sb S ⊕W R ⊕A L)
(is - ⇒d (?ts-A, ?m-A, ?share-A))
by (simp add: is_s) 

from i-bound' have i-bound": i < length ?ts-A
by simp

from valid-program-history [OF j-bound” ts_sj-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-sbj] )
  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), isj,
  v sj \{ (dom v sj – read-tmps (ys @ Read sb False a’ t v# zs)), ()Dj,
 acquired True ?take-sbj O_j,release ?take-sbj (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_sj-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-sbj])
  apply (simp only: split-suspends j [symmetric] suspends j)
  apply simp
done
from valid-reads [OF j-bound” ts_sj-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_sj-j j-bound”
  ts_sj-j reads-consis]
have reads-consis-m: reads-consistent True (acquired True ?take-sbj 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_sj-i
  ts_sj-j]
have outstanding-ref is-Write_s sb ∩ outstanding-ref 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-sbj O_j)
?m-A suspends_j.

**hence** reads-consistent ?m-A-ys: reads-consistent True (acquired True ?take-sb\_j \sigma_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 \theta_{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-program-hd-program: last-program (hd-program p_j sb\_j) ?take-sb\_j = (hd-program p_j suspends\_j)

proof
  from last-program have last-program p_j (?take-sb\_j@?drop-sb\_j) = p_j by simp
from last-program-hd-program-append'' [OF h-consis] this
have last-program (hd-program p_j suspends\_j) ?take-sb\_j = hd-program p_j suspends\_j
by (simp only: split-suspends\_j [symmetric] suspends\_j)
moreover
  have last-program (hd-program p_j (?take-sb\_j @ suspends\_j)) ?take-sb\_j = last-program (hd-program p_j suspends\_j) ?take-sb\_j
apply (simp only: split-suspends\_j [symmetric] suspends\_j)
by (rule last-program-hd-program-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

from history-consistent-appendD [OF valid-sops-take read-tmps-take-drop h-consis]

last-program-hd-program

611
have hist-consis': history-consistent θsbj (hd-prog p̂j suspendsj) suspendsj
  by (simp add: split-suspendsj [symmetric] suspendsj)
from reads-consistent-drop-volatile-writes-no-volatile-reads
  [OF reads-consis]
have no-vol-read: outstanding-refs is-volatile-Read̂sbys = {}
  by (auto simp add: outstanding-refs-append suspendsj [symmetric]
    split-suspendsj)
from flush-store-buffer-append [OF j-bound′′′′
    hist-consis′ [simplified split-suspendsj] valid-sops-drop distinct-read-tmps-drop
  [simplified split-suspendsj]
  no-volatile-Read̂sb-bo-volatile-reads-consistent [OF no-vol-read],
where
S=?share-A
do
obtain isj′ R̂j′ where
isj': instrs (Read̂sb False a′ t v # zs) @ iŝsbj =
  isj′ @ prog-instrs (Read̂sb False a′ t v # zs) and
steps-ys: (?ts-A, ?m-A, ?share-A) ⇒d*(?ts-A[j:= (last-prog (hd-prog p̂j (Ghost sb A′ L′ R′ W′# zs))) ys,
    isj′,
    θ̂sbj | (dom θ̂sbj − read-tmps (Read̂sb False a′ t v # zs)),(),
    D̂j ∨ outstanding-refs is-volatile-Writêsbys ≠ {}, acquired True ys (acquired
    True ?take-sb̂ j Ôj, R̂j′) ],
    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 iŝsbj ((ys @ [Read̂sb False a′ t v]) @ zs)
  by simp
from causal-program-history-suffix [OF this]
have cph': causal-program-history iŝsbj zs.
interpret causalj: causal-program-history iŝsbj zs by (rule cph')
from causalj.causal-program-history [of [], simplified, OF refl] isj'
obtain isj'' where
  isj' = Read False a′ t #isj'' and
  isj'': instrs zs @ iŝsbj = isj'' @ 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

612
have ts-ys-j: ?ts-ys!j=(last-prog (hd-prog p j (Read sb False a ’ t v# zs)) ys, isj’,
    \(v_{sbj} \vdash \) (dom \(v_{sbj}\) — read-tmps (Read sb False a ’ t v# 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 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’ / \in 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 sbj
proof –
{ assume a-notin: a’ / \in \(O_j\) \cup all-acquired sbj
from weak-sharing-consis [OF j-bound” ts sb-j]
have weak-sharing-consistent \(O_j\) sbj.
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 sbj
    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 sbj \(O_j\)]
    all-acquired-append [of ?take-sb j ?drop-sb j]
have a’ / \in acquired True (takeWhile (Not \circ is-volatile-Write sb) sbj) \(O_j\) \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 \(S \oplus W R \ominus A L\)) .

with a’-not-ro have False
by auto
}

613
thus \(?\)thesis by blast

qed

moreover
from A-unquired-by-others \([\text{rule-format}, \text{OF - neq-i-j}]\) ts\(_{sb-j}\) j-bound
have A \(\cap\) all-acquired sb\(_j\) = \{\}
by (auto simp add: Let-def)

moreover
from A-unownerd-by-others \([\text{rule-format}, \text{OF - neq-i-j}]\) ts\(_{sb-j}\) j-bound
have A \(\cap\) \(O\)_j = \{\}
by (auto simp add: Let-def dest: all-shared-acquired-in)

moreover note a\(^{\text{\textasciicircum}}\)-in
ultimately
show False
by auto
qed

}\?thesis
by (auto simp add: Let-def)
qed

have valid-own\(^{\text{\textasciicircum}}\): valid-ownership \(S\)\(_{sb}\) \(\prime\) \(ts\)\(_{sb}\) \(\prime\)

proof (intro-locales)

show outstanding-non-volatile-refs-owned-or-read-only \(S\)\(_{sb}\) \(\prime\) \(ts\)\(_{sb}\) \(\prime\)

proof –
from outstanding-non-volatile-refs-owned-or-read-only \([\text{OF i-bound ts}_{sb-i}]\)
have non-volatile-owned-or-read-only False \(S\)\(_{sb}\) \(O\)_\(sb\) (sb \(\oplus\) [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 \([\text{OF i-bound this}]\)
show \(?\)thesis by (simp add: ts\(_{sb}\) \(\prime\) sb\(_{j}\) \(O\)\(_{sb}\) \(\prime\) \(S\)\(_{sb}\) \(\prime\))

qed

next

show outstanding-volatile-writes-unowned-by-others ts\(_{sb}\) \(\prime\)

proof (unfold-locales)
fix i\(_{1}\) j p\(_{1}\) is\(_{1}\) \(O\)\(_{i}\) \(R\)\(_{1}\) \(D\)\(_{1}\) 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\(_{sb}\) \(\prime\)
assume j-bound: j < length ts\(_{sb}\) \(\prime\)
assume i\(_{1}\)-j: i\(_{1}\) \(\neq\) j
assume ts-i\(_{1}\): ts\(_{sb}\) \(\langle\)\(_{i_{1}}\) = (p\(_{1}\).is\(_{1}\).xs\(_{1}\).sb\(_{1}\).D\(_{1}\).O\(_{1}\).R\(_{1}\))
assume ts-j: ts\(_{sb}\) \(\langle\)\(_{j}\) = (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
with i\(_{1}\)-j have i-j: i\(_{1}\) \(\neq\) j
by simp

from j-bound have j-bound\(^{\text{\textasciicircum}}\): j < length ts\(_{sb}\)
by (simp add: ts\(_{sb}\) \(\prime\))

hence j-bound\(^{\text{\textasciicircum}}\): j < length (map owned ts\(_{sb}\))
by simp

614
from ts-j i-j have ts-j': ts_{sb}^[i-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 ' i-j ts_{sb}-i ts-j']
have (O_{j} \cup all-acquired sb_{j}) \cap outstanding-refs is-volatile-Write_{sb} sb_{1} = {}.
with ts-i \ True i-bound show ?thesis
by (clarsimp simp add: ts_{sb})

next
  case False
  note i1-i = this
  from i1-bound have i1-bound': i_{1} < length ts_{sb}
  by (simp add: ts_{sb})
  from ts-i1 False have ts-i1': ts_{sb}^[i1] = (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 i1-bound'
  have i1-bound''': i_{1} < length (map owned ts_{sb})
  by simp
  from outstanding-volatile-writes-unowned-by-others
  [OF i1-bound'' i-bound j-bound i1-j ts-i1'' ts-j']
  have (O_{sb} \cup all-acquired sb) \cap outstanding-refs is-volatile-Write_{sb} sb_{1} = {}.
  moreover
  from A-unused-by-others [rule-format, OF - False [symmetric]] False ts-i1 i1-bound
  have A \cap 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}')
  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} \cup all-acquired sb_{j}) \cap 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 \theta_n sb_n p_m is_m O_m R_m D_m \theta_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, \theta_n, sb_n, D_n, O_n, R_n)  
and mth: ts_{sb}'!m = (p_m, is_m, \theta_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 \text{all-acquired sb}_m) \cap 
\text{read-only-reads (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)_n) O_n) (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, is_n, \theta_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 \text{read-only-reads (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)_n) O_n) (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)_n} = 
\{\}  
by auto  
ultimately  
show ?thesis  
using True ts_{sb}'!nth' mth n-bound' m-bound'  
by (auto simp add: ts_{sb}' O_{sb}' sb' all-acquired-append)  
next  
case False  
note neq-m-i = this  
with m-bound nth i-bound have nth': ts_{sb}'!m = (p_m, is_m, \theta_m, 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']  
then show ?thesis  
using True neq-m-i ts_{sb}'!nth' m-bound' m-bound'  
apply (case-tac outstanding-refs (is-volatile-Write_{sb}) sb = \{\})  
apply (clarsimp simp add: outstanding-vol-write-take-drop-appends acquired-append read-only-reads-append ts_{sb}' sb' O_{sb}')+  
done  
next  

616
case False
  with n-bound nth i-bound have nth': ts_{sb}!n = (p_n, is_n, \theta_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
next
show ownership-distinct ts_{sb}'
proof -
  have \( \forall j < \text{length } ts_{sb}. \ i \neq j \rightarrow \)
    (let \((p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j) = ts_{sb}! j\) \in \((O_{sb} \cup \text{all-acquired } sb') \cap (O_j \cup \text{all-acquired } sb_j) = \{\}\))
  proof -
  \{ \text{fix } j \ p_j \ is_j \ R_j \ D_j \ \theta_j \ sb_j \}
  assume neq-i-j: i \neq j
  assume j-bound: j < \text{length } ts_{sb}
  assume ts_{sb-b-j}: ts_{sb}! j = (p_j, is_j, \theta_j, sb_j, D_j, O_j, R_j)
  have \((O_{sb} \cup \text{all-acquired } sb') \cap (O_j \cup \text{all-acquired } sb_j) = \{\}\)
  proof -
  \{ \text{fix } a' \}
  assume a'-in-i: a' \in (O_{sb} \cup \text{all-acquired } sb')
  assume a'-in-j: a' \in (O_j \cup \text{all-acquired } sb_j)
  have False
  proof -
  from a'-in-i have a' \in (O_{sb} \cup \text{all-acquired } sb) \lor a' \in A
  by (simp add: sb' all-acquired-append)
  then show False
  proof
  assume a' \in (O_{sb} \cup \text{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 < \text{length } (\text{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 \cap \text{acquired } True (\text{takeWhile } (\text{Not } is-volatile-Write_{sb}) sb_j) O_j = \{\}\ and
    A \cap \text{all-shared } (\text{takeWhile } (\text{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 \cap \text{all-acquired } sb_j = \{\}
  by (auto simp add: Let-def)
by auto
  ultimately
  show \textit{thesis}
using \textit{a}′-in-j
by (auto dest: all-shared-acquired-in)
qed
qed

\textbf{then show} \textit{thesis} by auto
  qed

\textbf{then show} \textit{thesis} by (fastforce simp add: Let-def)
qed

from ownership-distinct-nth-update [OF i-bound ts\,sb\,-i this]
show \textit{thesis} by (simp add: ts\,sb′/O\,sb′/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 \(p\sb\)\,sb) \(sb\),
with valid-write-sops [OF i-bound ts\,sb\,-i]
  valid-implies-valid-prog-hd [OF i-bound ts\,sb\,-i valid]
have history-consistent \(\theta\)\,sb (hd-prog \(p\sb\) (\(sb\@[\text{Ghost}_{sb}\ A\ L\ R\ W]\)))
  (\(sb\@[\text{Ghost}_{sb}\ A\ L\ R\ W]\))
apply --
apply (rule history-consistent-appendI)
apply (auto simp add: hd-prog-append-Ghost\sb)
done
from valid-history-nth-update [OF i-bound this]
show \textit{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 \textit{False} \(O\sb\,m\sb\,sb\).
from reads-consistent-snec-Ghost\sb [OF this]
have reads-consistent \textit{False} \(O\sb\,m\sb\,sb\ (\text{sb}\@[\text{Ghost}_{sb}\ A\ L\ R\ W])).
from valid-reads-nth-update [OF i-bound this]
show \textit{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\ (\text{sb}\@[\text{Ghost}_{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: $ts_{sb}'$, $sb'$ $S_{sb}'$)
next
from sharing-consis [OF i-bound $ts_{sb}$]
have consis': sharing-consistent $S_{sb}$ $O_{sb}$ $sb$.
from A-shared-owned
have $A \subseteq $ dom (share $?drop-sb$ $S$) $\cup$ acquired True $sb$ $O_{sb}$
by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write$sb$)
moreover have dom (share $?drop-sb$ $S$) $\subseteq $ dom $S$ $\cup$ dom (share sb $S_{sb}$)
proof
fix $a'$
assume $a'$-in: $a' \in$ dom (share $?drop-sb$ $S$)
from share-unshared-in [OF $a'$-in]
show $a' \in$ dom $S$ $\cup$ dom (share sb $S_{sb}$)
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 $S_{sb}$)
by auto
thus $?thesis$
by simp
next
assume $a' \in$ dom $S$ $\land$ $a' \notin$ all-unshared $?drop-sb$
thus $?thesis$ by auto
qed
qed
ultimately
have A-subset: $A \subseteq $ dom $S$ $\cup$ dom (share sb $S_{sb}$) $\cup$ acquired True $sb$ $O_{sb}$
by auto
have $A \subseteq $ dom (share sb $S_{sb}$) $\cup$ acquired True $sb$ $O_{sb}$
proof
{ fix $x$
assume $x$-$A$: $x \in$ $A$
have $x \in$ dom (share sb $S_{sb}$) $\cup$ acquired True $sb$ $O_{sb}$
proof
{ assume $x \in$ 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

619
qed
with consis′ L-subset A-R R-acq
have sharing-consistent $S_{sb}$ $O_{sb}$ ($sb @ [\text{Ghost}_{sb} A L R W]$)
  by (simp add: sharing-consistent-append acquired-takeWhile-non-volatile-Write$_{sb}$)
from sharing-consis-nth-update [OF i-bound this]
show sharing-consis $S_{sb}′$ $ts_{sb}′$
  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 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-outstanding-write-to-read-only-memory $S_{sb}′$ $ts_{sb}′$
  by (simp add: $S_{sb}′$ $ts_{sb}′$ $sb′$)
qed

have tmpts-distinct′: tmpts-distinct $ts_{sb}′$
  proof (intro-locales)
from load-tmpts-distinct [OF i-bound $ts_{sb}′$-i]
have distinct-load-tmpts $is_{sb}′$ by (simp add: $is_{sb}′$)
from load-tmpts-distinct-nth-update [OF i-bound this]
show load-tmpts-distinct $ts_{sb}′$ by (simp add: $ts_{sb}′$)
next
from read-tmpts-distinct [OF i-bound $ts_{sb}′$-i]
have distinct-read-tmpts ($sb @ [\text{Ghost}_{sb} A L R W]$)
  by (auto simp add: distinct-read-tmpts-append)
from read-tmpts-distinct-nth-update [OF i-bound this]
show read-tmpts-distinct $ts_{sb}′$ by (simp add: $ts_{sb}′$ $sb′$)
next
from load-tmpts-read-tmpts-distinct [OF i-bound $ts_{sb}′$-i]
have load-tmpts $is_{sb}′$ $\cap$ read-tmpts ($sb @ [\text{Ghost}_{sb} A L R W]$) ={} 
  by (auto simp add: read-tmpts-append $is_{sb}′$)
from load-tmpts-read-tmpts-distinct-nth-update [OF i-bound this]
show load-tmpts-read-tmpts-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]

620
obtain 
valid-store-sops' : \( \forall \text{sop} \in \text{store-sops} \) is \( \text{sop} \)
by (auto simp add: is sb)
from valid-write-sops [OF i-bound ts sb-i]
have valid-write-sops' : \( \forall \text{sop} \in \text{write-sops (sb@ [Ghost sb A L R W])} \)
valid-sop
by (auto simp add: write-sops-append)
from valid-sops-nth-update [OF i-bound valid-write-sops'
valid-store-sops']
show \(?\text{thesis}\) by (simp add: ts sb'

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 \( \emptyset \) sb') is sb'
by (auto simp add: is sb sb')
from load-tmps-write-tmps-distinct [OF i-bound ts sb-i]
have load-tmps is sb' \( \cap \) \( \emptyset \) (fst ' write-sops (sb@ [Ghost 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 \(?\text{thesis}\) by (simp add: ts sb 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' \( \cap \) \( \emptyset \) sb = \{\}
by (auto simp add: is sb)
from load-tmps-fresh-nth-update [OF i-bound this]
show \(?\text{thesis}\) by (simp add: ts sb sb'

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 \text{outstanding-refs is-volatile-Write sb (sb@[Ghost sb A L R W])} = \{\}
by (auto simp add: outstanding-refs-append)
from enough-flushs-nth-update [OF i-bound this]
show \(?\text{thesis}\) by (simp add: ts sb sb' D sb')

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 .
than have causal': causal-program-history is sb' (sb@[Ghost sb A L R W])
by (auto simp: causal-program-history-Ghost is sb)

621
from valid-last-prog [OF i-bound \text{ts}_{sb} \cdot i] 

have \text{last-prog} \ p_{sb} \ \text{sb} = p_{sb}.

hence \text{last-prog} \ p_{sb} \ \text{sb} \ (\text{sb} @ [\text{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 \cdot this]

show \ ?thesis

by (simp add: \text{ts}_{sb} \cdot sb)

qed

show \ ?thesis

proof (cases outstanding-refs is-volatile-Write_{sb} \ \text{sb} = \{\})

case True

have \ flush-all: takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \ \text{sb} = \text{sb}

by (auto simp add: outstanding-refs-conv)

from True have \ suspend-nothing: dropWhile (Not \circ \text{is-volatile-Write}_{sb}) \ \text{sb} = []

by (auto simp add: outstanding-refs-conv)

hence suspends-empty: suspends = []

by (simp add: suspends)

from suspends-empty is-sim have \ is = \text{Ghost} A L R W\# is_{sb} \cdot'

by (simp add: is_{sb})

with suspends-empty ts-i

have ts-i: tsli = (p_{sb}, \text{Ghost} A L R W\# is_{sb} \cdot',

\ \text{\ddot{d}}_{sb}(), D, \text{acquired True} \ ?\text{take-sb} \ \text{O}_{sb} \ \text{release} \ ?\text{take-sb} \ \text{(dom} \ S_{sb}) \ \text{R}_{sb})

by simp

from direct-memop-step.Ghost

have (\text{Ghost} A L R W\# is_{sb} \cdot,',

\ \text{\ddot{d}}_{sb}(), m, D, \text{acquired True} \ ?\text{take-sb} \ \text{O}_{sb},

\ \text{release} \ ?\text{take-sb} \ \text{(dom} S_{sb}) \ \text{R}_{sb}, S) \rightarrow

(is_{sb} \cdot',

\ \text{\ddot{d}}_{sb}(), m, D, \text{acquired True} \ ?\text{take-sb} \ \text{O}_{sb} \cup A - R,

\ \text{augment-rels} \ (\text{dom} S) \ R \ \text{(release} \ ?\text{take-sb} \ \text{(dom} S_{sb}) \ \text{R}_{sb}),

S \oplus W R \ominus A L).

from direct-computation.concurrent-step.Memop [OF i-bound \cdot ts-i this]

have (ts, m, S) \Rightarrow_d

(ts|i := (p_{sb}, is_{sb} \cdot'),

\ \text{\ddot{d}}_{sb}(), D, \text{acquired True} \ ?\text{take-sb} \ \text{O}_{sb} \cup A - R,

\ \text{augment-rels} \ (\text{dom} S) \ R \ \text{(release} \ ?\text{take-sb} \ \text{(dom} S_{sb}) \ \text{R}_{sb})),

m,S \oplus W R \ominus A L).

moreover

from suspend-nothing

have suspend-nothing\cdot: \ (\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}' = []

by (simp add: sb')

622
have all-shared-A: \( \forall j \ p \ is \ O \ R \ D \ \emptyset \ sb. \ j < length ts_{sb} \rightarrow i \neq j \rightarrow \\
\text{ts}_{sb} \ ! \ j = (p, \ is, \ \emptyset, \ sb, \ D, \ O, R) \rightarrow \\
\text{all-shared} \ (\text{takeWhile} \ (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ sb) \cap \ A = \{\}

proof

\{ 
fix \ j p j_i O_j R_j D_j \ \emptyset_j sb_j x 
assume j-bound: \ j < length ts_{sb} 
assume neq-i-j: i \neq j 
assume jth: ts_{sb}!j = (p_j, is_j, \emptyset_j, sb_j, D_j, O_j, R_j) 
assume x-shared: x \in \text{all-shared} \ (\text{takeWhile} \ (\text{Not} \ o \ \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 all-acquired sb_j \cup O_j, 
moreover have all-shared \ (\text{takeWhile} \ (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ sb_j) \subseteq all-shared sb_j 
using all-shared-append [of \ (\text{takeWhile} \ (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ sb_j) \\
(\text{dropWhile} \ (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ sb_j)] 
by auto 
moreover 
from A-unacquired-by-others [rule-format, OF - neq-i-j] jth j-bound 
have A \cap all-acquired sb_j = \{\} by auto 
moreover 
from A-unowned-by-others [rule-format, OF - neq-i-j] 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 
\}

thus \( ?\)thesis by blast 
qed

hence all-shared-L: \( \forall j \ p \ is \ O \ R \ D \ \emptyset \ sb. \ j < length ts_{sb} \rightarrow i \neq j \rightarrow \\
\text{ts}_{sb} \ ! \ j = (p, \ is, \ \emptyset, \ sb, \ D, \ O, R) \rightarrow \\
\text{all-shared} \ (\text{takeWhile} \ (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ sb) \cap L = \{\}
using L-subset by blast
have all-shared-A: \( \forall j \ p \ is \ O \ R \ D \ \emptyset \ sb. \ j < length ts_{sb} \rightarrow i \neq j \rightarrow \\
\text{ts}_{sb} \ ! \ j = (p, \ is, \ \emptyset, \ sb, \ D, \ O, R) \rightarrow 

all-shared \((\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{ab}) \text{sb}) \cap A = \{\}\)

proof –

\(\{\)

\text{fix } j \ p_j \ is_j \ O_j \ R_j \ D_j \ \varnothing_j \ 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_j,\text{sb}_j,D_j,O_j,R_j)\)

\text{assume } neq-i-j: i \neq j

\text{assume } x\text{-shared}: x \in \text{all-shared} \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb)\)

\text{assume } x\text{-A}: x \in A

\text{have } False

proof –

\text{from all-shared-acquired-or-owned [OF sharing-consis [OF j\text{-bound j\text{th}]]]

\text{have all-shared sb}_j \subseteq \text{all-acquired sb}_j \cup O_j\).

moreover \text{have all-shared} \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \subseteq \text{all-shared

sb}_j\)

using all-shared-append \ [of \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb)\]

\ (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb)\)]

by auto

moreover

\text{from A-unacquired-by-others [rule-format, OF - neq-i-j] j\text{th j\text{-bound

\text{have A} \cap \text{all-acquired sb}_j = \{\} by auto

moreover

\text{from A-unowned-by-others [rule-format, OF - neq-i-j] j\text{th j\text{-bound

\text{have A} \cap \text{O}_j = \{\} \text{ by (auto dest: all-shared-acquired-in

ultimately

\text{show False

using x\text{-A x\text{-shared}

by blast

qed

}\}

\text{thus } ?\text{thesis by blast

qed

\text{hence all-shared-L: } \forall j \ p \ is \ O \ R \ D \ \varnothing \ sb. \ j < \text{length } ts_{sb} \rightarrow i \neq j \rightarrow

\text{ts}_{sb} ! j = (p, \text{is}, \ \varnothing, \ \text{sb}, \ D, O, R) \rightarrow

\text{all-shared} \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \cap L = \{\}

using L\text{-subset by blast

\text{have all-unshared-R: } \forall j \ p \ is \ O \ R \ D \ \varnothing \ sb. \ j < \text{length } ts_{sb} \rightarrow i \neq j \rightarrow

\text{ts}_{sb} ! j = (p, \text{is}, \ \varnothing, \ \text{sb}, \ D, O, R) \rightarrow

\text{all-unshared} \ (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \cap R = \{\}

proof –

\(\{\)

\text{fix } j \ p_j \ is_j \ O_j \ R_j \ D_j \ \varnothing_j \ sb_j \ x\)

\text{assume } j\text{-bound}: j < \text{length } ts_{sb}\)

\text{assume } neq-i-j: i \neq j

624
assume jth: ts_{sb}!j = (p_j, is_j, θ_j, sb_j, D_j, O_j, R_j)
assume x-unshared: x ∈ all-unshared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j)
assume x-R: x ∈ R
have False
proof –
  from unshared-acquired-or-owned [OF sharing-consis [OF j-bound jth]]
  have all-unshared sb_j ⊆ all-acquired sb_j ∪ O_j,

moreover have all-unshared (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j) ⊆ all-unshared sb_j
using all-unshared-append [of (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j)
  (dropWhile (Not ◦ 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-unshared acquired-all-acquired [of True sb O_{sb}]
  by blast
qed

thus ?thesis by blast
qed

have all-acquired-R: ∀ p is O R D sb. j < length ts_{sb} → i ≠ j →
  ts_{sb}! j = (p, is, D, O, R) →
  all-acquired (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) ∩ R = {}
proof –
{
  fix j p_j is_j O_j R_j D_j sb_j x
  assume j-bound: j < length ts_{sb}
  assume jth: ts_{sb}!j = (p_j, is_j, θ_j, sb_j, D_j, O_j, R_j)
  assume neq-i-j: i ≠ j
  assume x-acq: x ∈ all-acquired (takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j)
  assume x-R: x ∈ R
  have False
proof –
  from x-acq have x ∈ all-acquired sb_j
  using all-acquired-append [of takeWhile (Not ◦ is-volatile-Write_{sb}) sb_j
dropWhile (Not ◦ 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 acquired-all-acquired [of True sb O_{sb}]
by blast

625
qed

thus \textit{thesis} by blast

qed

have all-shared-R: \( \forall \ j \ p \ \text{is} \ \varnothing \ \text{sb}, \ j < \text{length} \ ts_{sb} \rightarrow i \neq j \rightarrow \)
\[ ts_{sb} \ ! j = (p, \text{is}, \varnothing, \text{sb}, D, O, R) \rightarrow \]
all-shared \((\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}) \cap R = \{\}\)

proof -
{
fix \ j \ p_j \text{is}_{j} O_j R_j D_j \ 0_j \text{sb}_j x
assume j-bound: \( j < \text{length} \ ts_{sb} \)
assume jth: \( ts_{sb}[j] = (p_j, \text{is}_{j}, 0_j, \text{sb}_j, D_j, O_j, R_j) \)
assume neq-i-j: \( i \neq j \)
assume x-shared: \( x \in \text{all-shared} \ (\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}) \)
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 \( \text{sb}_j \subseteq \text{all-acquired} \) \( \text{sb}_j \cup O_j \).

moreover have all-shared \((\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}) \subseteq \text{all-shared} \ \text{sb}_j \)
using all-shared-append [of \((\text{takeWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb}) \)]
(\(\text{dropWhile} (\text{Not} \ o \ \text{is-volatile-Write}_{sb}) \ \text{sb})\)]
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 \( \text{sb} \) \( O_{sb} \)]
by blast
qed

thus \textit{thesis} by blast

qed

note share-commute =
share-all-until-volatile-write-append-Ghost_{sb} [OF True \( \text{ownership-distinct} \) \( ts_{sb} \)]
(\(\text{sharing-consis} \ S_{sb} ts_{sb} \))
i-bound \( ts_{sb}\)-i all-shared-L all-shared-A all-acquired-R all-unshared-R all-shared-R

from \( D \)
have \( D': \ D_{sb} = (D \lor \text{outstanding-refs} \ \text{is-volatile-Write}_{sb} \ (\text{sb}@[\text{Ghost}_{sb} A L R W]) \neq \{\}) \)
by (auto simp: outstanding-refs-append)

have \( \forall \ a \in R. \ (a \in (\text{dom} \ (\text{share} \ \text{sb} \ S_{sb}))) = (a \in \text{dom} \ S) \)
proof -
{

626
\textbf{fix} a
\textbf{assume} a-R: \(a \in \mathbb{R}\)
\textbf{have} \((a \in (\text{dom } (\text{share sb } S_{sb}))) \) = \((a \in \text{dom } S)\)
\textbf{proof} –
\textbf{from} a-R R-acq acquired-all-acquired \([\text{of True sb } O_{sb}]\)
\textbf{have} \(a \in O_{sb} \cup \text{all-acquired sb}\)
\textbf{by} auto

\textbf{from} \text{share-all-until-volatile-write-thread-local}' \text{[OF ownership-distinct-ts}_{sb}\text{ sharing-consis-ts}_{sb}\text{ i-bound ts}_{sb}-i\text{ this]} \text{ suspend-nothing}
\textbf{show} \ ?thesis \textbf{by} (auto simp add: domIff \(S\))
\textbf{qed}

\textbf{then show} \ ?thesis \textbf{by} auto
\textbf{qed}
\textbf{from} \text{augment-rels-shared-exchange} \text{[OF this]}
\textbf{have} rel-commute:
\textbf{augment-rels} \((\text{dom } S)\) R \((\text{release sb } (\text{dom } S_{sb}) \ R_{sb})\) =
\text{release} \((\text{sb @ } \text{Ghost}_{sb} A L R W)\) \((\text{dom } S_{sb}')\) \(R_{sb}\)
\textbf{by} (clarsimp simp add: release-append \(S_{sb}'\))

\textbf{have} \((ts_{sb}',m_{sb},S_{sb}')\sim\)
\((ts[i] := (p_{sb}-i\_sb'),\)
\(\partial_{sb}(), D, \text{ acquired True ?take-sb } O_{sb} \cup A - R,\)
\text{augment-rels} \((\text{dom } S)\) R \((\text{release ?take-sb } (\text{dom } S_{sb}) \ R_{sb}))\],
\(m,S \oplus W R \ominus A L)\)
\textbf{apply} (rule sim-config.intros)
\textbf{apply} (simp add: m ts_{sb} O_{sb} sb' \(\partial_{sb}'\))
\textbf{flush-all-until-volatile-write-append-Ghost-commute} \text{[OF i-bound ts}_{sb}-i\])
\textbf{apply} (clarsimp simp add: S S_{sb}' ts_{sb}' sb' O_{sb}' \(\partial_{sb}'\) share-commute)
\textbf{using} leq
\textbf{apply} (simp add: ts_{sb}')
\textbf{using} i-bound i-bound't ts-sim ts-i True \(D'\)
\textbf{apply} (clarsimp simp add: Let-def nth-list-update
\text{outstanding-refs-conv} ts_{sb}' O_{sb} R_{sb}' S_{sb}' \(\partial_{sb}'\) sb' \(D_{sb}'\) \text{ suspend-nothing} \text{flush-all}
\text{rel-commute
\text{acquired-append split: if-split-asm)}
\textbf{done}

\textbf{ultimately show} \ ?thesis
\textbf{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}' R_{sb}'
\textbf{by} auto
\textbf{next}
case False

\textbf{then obtain} r \textbf{where} r-in: \(r \in \text{set sb}\) \textbf{and} volatile-r: is-volatile-Write_{sb} r

627
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''
  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)
  apply (simp add: m flush-all-until-volatile-write-append-Ghost-commute [OF i-bound ts_{sb'-i} ts_{sb'} O_{sb'} d_{sb'} sb'])
  apply (clarsimp simp add: S_{sb'} ts_{sb'} sb' O_{sb'} d_{sb'} share-commute)
  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)
read-tmps-append suspends
prog-instrs-append-Ghost_{sb} instrs-append-Ghost_{sb}
drop is_{sb} ts_{sb} \cdot sb' O_{sb} \cdot R_{sb} \cdot S_{sb} \cdot \partial_{sb} \cdot D_{sb} \cdot takeWhile-app split: if-split-asm)
done
ultimately show \(\text{thesis}\)
using valid-own valid-hist valid-reads valid-sharing tmps-distinct valid-dd
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} \varnothing_{sb} sb D_{sb} O_{sb} R_{sb} S_{sb} θ_{sb} D
then obtain
ts_{sb} : ts_{sb} = ts_{sb}! i = (p_{sb}, is_{sb}, \varnothing_{sb}, sb, D_{sb}, O_{sb}, R_{sb}) and
flush: (m_{sb},sb,O_{sb},R_{sb},S_{sb}) \rightarrow_f
(by auto)

from sim obtain
m = flush-all-until-volatile-write ts_{sb} m_{sb} and
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}, \varnothing, sb, D_{sb}, O_{sb}, R) = ts_{sb}! i;
suspends = dropWhile (Not \circ is-volatile-Write sb)
in \exists D. instrs suspends @ is_{sb} = is @ prog-instrs suspends \land
D_{sb} = (D \land outstanding-refs is-volatile-Write_{sb} sb \neq \{} ) \land
\exists i = (\text{hd-prog p suspends},
is, \varnothing, (\text{dom } \varnothing - \text{read-tmps suspends}), (\),
\text{D},
\text{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 \circ is-volatile-Write_{sb} sb @) dropWhile (Not \circ is-volatile-Write_{sb} sb)
(is sb = ?take-sb@?drop-sb)
by simp

from ts-sim [rule-format, OF i-bound] ts_{sb}! obtain suspends is D where
suspends: \texttt{suspends} = \texttt{dropWhile (Not \circ \texttt{is-volatile-Write}\_sb)} \_sb \texttt{and} \\ is-sim: \texttt{instrs suspends @ i}_{\texttt{s}_b} = \texttt{is @ prog-instrs suspends and} \\ = (D \lor \texttt{outstanding-ref}\_\texttt{is-volatile-Write}\_\texttt{sb} \neq \emptyset) \texttt{and} \\ ts-i: \texttt{ts} \_i = \\
\texttt{(hd-prog p}_{\texttt{sb}} \texttt{suspends, is,} \\
\texttt{\_sb \_i \_v (dom } \texttt{\_sb \_i \_v - read-tmps suspends, )}, D, \texttt{acquired True ?take-sb } O_{\texttt{sb}}, \\
\texttt{release ?take-sb (dom } S_{\texttt{sb}}) R_{\texttt{sb}}) \\
\texttt{by (auto simp add: Let-def)} \\
\texttt{from flush-step-preserves-valid [OF i-bound ts}_{\texttt{sb}} \_i \texttt{flush valid} \\
\texttt{have valid’ : valid ts}_{\texttt{sb}}’ \\
\texttt{by (simp add: ts}_{\texttt{sb}}’) \\
\texttt{from flush obtain r where sb: sb=r\#sb’} \\
\texttt{by (cases) auto} \\
\texttt{from valid-history [OF i-bound ts}_{\texttt{sb}} \_i] \\
\texttt{have history-consistent } \_sb \_v (hd-prog p}_{\texttt{sb}} \texttt{sb} \texttt{sb,} \\
\texttt{then} \\
\texttt{have hist-consis’: history-consistent } \_sb \_v (hd-prog p}_{\texttt{sb}} \texttt{sb’} \texttt{sb’} \\
\texttt{by (auto simp add: sb intro: history-consistent-hd-prog} \\
\texttt{split: memref.splits option.splits) \\
\texttt{from valid-history-nth-update [OF i-bound this] \\
\texttt{have valid-hist’: valid-history program-step ts}_{\texttt{sb}}’ \texttt{by (simp add: ts}_{\texttt{sb}}’) \\
\texttt{from dist-sb’ : distinct-read-tmps sb’} \\
\texttt{by (simp add: sb split: memref.splits)} \\
\texttt{have tmps-distinct’: tmps-distinct ts}_{\texttt{sb}}’ \\
\texttt{proof (intro-locales)} \\
\texttt{from load-tmps-distinct [OF i-bound ts}_{\texttt{sb}} \_i] \\
\texttt{have distinct-load-tmps i}_{\texttt{sb}}. \\
\texttt{next} \\
\texttt{from load-tmps-distinct-nth-update [OF i-bound this]} \\
\texttt{show load-tmps-distinct ts}_{\texttt{sb}}’ \\
\texttt{by (simp add: ts}_{\texttt{sb}}’) \\
\texttt{next} \\
\texttt{from load-tmps-read-tmps-distinct [OF i-bound dist-sb’]} \\
\texttt{show load-tmps-distinct ts}_{\texttt{sb}}’ \\
\texttt{by (simp add: ts}_{\texttt{sb}}’) \\
\texttt{next} \\
\texttt{from load-tmps-read-tmps-distinct [OF i-bound ts}_{\texttt{sb}} \_i] \\
\texttt{have load-tmps is}_{\texttt{sb}} \cap \texttt{read-tmps sb’} = \emptyset \\
\texttt{by (auto simp add: sb split: memref.splits)} \\
\texttt{from load-tmps-read-tmps-distinct-nth-update [OF i-bound this]} \\
\texttt{show load-tmps-read-tmps-distinct ts}_{\texttt{sb}}’ \texttt{by (simp add: ts}_{\texttt{sb}}’) \\
\texttt{qed} \\
\texttt{from load-tmps-write-tmps-distinct [OF i-bound ts}_{\texttt{sb}} \_i]
have load-tmps \( i_{sb} \cap \bigcup (\text{fst ', write-sops sb'}) = \{\} \)
  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 splits)
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} \cap \text{dom } \theta_{sb} = \{\},
  from load-tmps-fresh-nth-update [OF i-bound this]
  show \(?thesis\) by (simp add: ts sb sb')
qed

have enough-flushs': enough-flushs ts_{sb}'
proof
  from clean-no-outstanding-volatile-Write sb [OF i-bound ts_{sb,i}]
  have \( \neg D_{sb} \longrightarrow \text{outstanding-refs is-volatile-Write}_{sb} \ sb' = \{\}\)
  by (auto simp add: sb split: 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: \( \neg \text{volatile } \rightarrow a \in \mathcal{O}_{sb} \)
  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 \( \in \mathcal{O}_{sb} \)
  by (simp add: sb Write_{sb})
  thus \(?thesis\) by simp
  qed

  have a-unowned-by-others:
  \( \forall j < \text{length } ts_{sb}, i \neq j \longrightarrow (s_{j,r}, -sb_{j,r}, \mathcal{O}_{j,r}) = ts_{sb} ! j \in \)

  631
\( a \not\in O_j \cup \text{all-acquired sb}_j \)

**proof** (unfold Let-def, clarify del: notI)

fix \( j \ p_j \ i_{sbj} O_j R_j \theta_j sb_j \)

**assume** \( j \)-bound: \( j < \text{length ts}_{sb} \)

**assume** neq: \( i \not= j \)

**assume** ts\( j \): \( ts_{sb} ! j = (p_j,i_{sbj},\theta_j,\theta_{sb},D_{sbj},O_j,R_j) \)

**show** \( a \not\in O_j \cup \text{all-acquired sb}_j \)

**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

from valid-reads [OF i-bound ts\( _{sb} \)-i]

**have** reads-consis: reads-consistent False \( O_{sb} m_{sb} sb \).


\{ 

fix \( j \)

fix \( p_j i_{sbj} O_j R_{sbj} \theta_j sb_j \)

**assume** \( j \)-bound: \( j < \text{length ts}_{sb} \)

**assume** ts\( _{sb} \)-j: \( ts_{sb} ! j = (p_j,i_{sbj},\theta_j,\theta_{sb},D_{sbj},O_j,R_j) \)

**assume** neq-i-j: \( i \not= j \)

**have** \( a \not\in \text{outstanding-refs is-Write}_{sb} \) (takeWhile (Not o is-volatile-Write\( _{sb} \)) sb\( j \))

**proof**

**assume** \( a \in \text{outstanding-refs is-Write}_{sb} \) (takeWhile (Not o is-volatile-Write\( _{sb} \)) sb\( j \))

**hence** \( a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \) (takeWhile (Not o is-volatile-Write\( _{sb} \)) sb\( j \))

**by** (simp add: outstanding-refs-is-non-volatile-Write\( _{sb} \)-takeWhile-conv)

**hence** \( a \in \text{outstanding-refs is-non-volatile-Write}_{sb} \) sb\( j \)

**using** outstanding-refs-append [of - (takeWhile (Not o is-volatile-Write\( _{sb} \)) sb\( j \)) (dropWhile (Not o is-volatile-Write\( _{sb} \)) sb\( j \))]

**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\( _{sb} \)-i]]

**have** \( a \in O_j \cup \text{all-acquired sb}_j \)

**by** auto

**with** a-unowned-by-others [rule-format, OF j-bound neq-i-j] ts\( _{sb} \)-j

**show** False

632
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, sb_j, \cdot, \cdot, \cdot) = ts_{sb}!j \ \text{in a } \notin \text{outstanding-refs is-Write}_{sb} (\text{takeWhile (Not } \circ \text{is-volatile-Write}_{sb}) \ sb_j))
by (fastforce simp add: Let-def)

obtain D f where sop: sop=(D,f) by (cases sop) auto
from valid-history [OF i-bound ts_{sb}-i] sop sb Write_{sb}
obtain D-tmps: D \subseteq \text{dom } \vartheta_{sb} \ \text{and } f-v: f \vartheta_{sb} = v \ \text{and}
D-sb': D \cap \text{read-tmps } sb' = \{\}
by auto
let ?\theta = (\vartheta_{sb} | (\text{dom } \vartheta_{sb} - \text{read-tmps } sb'))
from D-tmps D-sb'
have D-tmps': D \subseteq \text{dom } ?\theta
by auto
from valid-write-sops [OF i-bound ts_{sb}-i, rule-format, of sop]
have valid-sop sop
by (auto simp add: sb Write_{sb})
from this [simplified sop]
interpret valid-sop (D,f).
from D-tmps D-sb'
have ((\text{dom } \vartheta_{sb} - \text{read-tmps } sb') \cap D) = D
by blast
with valid-sop [OF refl D-tmps] valid-sop [OF refl D-tmps'] f-v
have f-v': f ?\theta = v
by auto

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 Write_{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 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 and
  S$_{sb}'$: S$_{sb}'$ = S$_{sb}$ $\oplus_W$ R $\cup_A$ L and
  R$_{sb}'$: R$_{sb}'$ = Map.empty
by cases (auto simp add: sb)

from sharing-consis [OF i-bound ts$_{sb}$-i]
obtain
  A-shared-owned: A $\subseteq$ 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 Write$_{sb}$ volatile)

from sb Write$_{sb}$ True have take-empty: takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb = []
by (auto simp add: outstanding-refs-conv)

from sb Write$_{sb}$ True have suspend-all: dropWhile (Not $\circ$ 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$_{sb}$-i]
interpret causal-program-history is$_{sb}$ sb .
from valid-last-prog [OF i-bound ts$_{sb}$-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$_{sb}$ = 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.intros)
apply (auto simp add: sb Write)
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)

let ?ts' = ts[i := (hd-prog p sb 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 volatile)

{ fix j
  fix p j is sbj O j R j D sbj θ j sbj
  assume j-bound: j < length ts sb
  assume ts sb-j: ts sbj = (p j, is sbj, θ j, sbj, 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-sb j = takeWhile (Not ◦ is-volatile-Write sb) sb j
  let ?drop-sb j = dropWhile (Not ◦ is-volatile-Write sb) sb j
  assume a-in: a ∈ unforwarded-non-volatile-reads ?drop-sb j {}
proof
  from a-unowned-by-others [rule-format, OF j-bound neq-i-j] ts sb-j
  obtain a-unowned: a /∈ O j and a-unacq: a /∈ all-acquired sb j
  by auto
  with all-acquired-append [of ?take-sb j ?drop-sb j]
  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: )

note nvo-j = outstanding-non-volatile-refrs-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 .

635
\textbf{note} \( \text{consis-j} = \text{sharing-consis [OF } j \text{-bound ts}_{sb-j} \) \\
\textbf{with} \( \text{sharing-consistent-append} \ [\text{of } S_{sb} O_j ?\text{take-sb} \_j ?\text{drop-sb} \_j] \) \\
\textbf{obtain} \( \text{consis-take-j} : \text{sharing-consistent} S_{sb} O_j ?\text{take-sb} \_j \) and \( \text{consis-drop-j} : \text{sharing-consistent} (\text{share } ?\text{take-sb} \_j S_{sb}) \) \\
\quad \text{(acquired True } ?\text{take-sb} \_j O_j \) ?\text{drop-sb} \_j \) \\
\quad \text{by auto} \\
\textbf{from} \ \text{in-unforwarded-non-volatile-reads-non-volatile-Read}_{sb} \ [\text{OF } a \text{-in}] \\
\textbf{have} \ a \text{-in}' : a \in \text{outstanding-refs is-non-volatile-Read}_{sb} ?\text{drop-sb} \_j. \\
\textbf{note} \( \text{reads-consis-j} = \text{valid-reads [OF } j \text{-bound ts}_{sb-j} \) \\
\textbf{from} \ \text{reads-consistent-drop} \ [\text{OF this}] \\
\textbf{have} \ \text{reads-consis-drop-j} : \text{reads-consistent True} (\text{acquired True } ?\text{take-sb} \_j O_j) (\text{flush } ?\text{take-sb} \_j m_{sb}) ?\text{drop-sb} \_j. \\
\textbf{from} \ \text{read-only-share-all-shared [of } a ?\text{take-sb} \_j S_{sb} \) \text{a-not-ro} \\
\quad \text{all-shared-acquired-or-owned [OF consis-take-j]} \\
\quad \text{all-acquired-append [of } ?\text{take-sb} \_j ?\text{drop-sb} \_j \) \text{a-unowned a-unacq} \\
\textbf{have} \ \text{a-not-ro-j} : a \notin \text{read-only} (\text{share } ?\text{take-sb} \_j S_{sb}) \\
\quad \text{by auto} \\
\textbf{from} \ \text{ts-sim [rule-format, OF } j \text{-bound ts}_{sb-j} \) \text{j-bound} \\
\textbf{obtain} \ \text{suspends}_{j} : \text{suspends}_{j} = ?\text{drop-sb} \_j \) and \\
\text{is}_{j} \text{ instrs suspends}_{j} @ is_{sbj} = is_{j} @ \text{prog-instrs suspends}_{j} \) and \\
\quad \text{D}_{j} : D_{sbj} = (D_{j} \lor \text{outstanding-refs is-volatile-Write}_{sb} \) \text{sb}_{j} \neq \text{\{}\} \) and \\
\quad \text{ts}_{j} : ts_{j} = (\text{hd-prog } p_{j} \text{ suspends}_{j}, is_{j}, \\
\quad \) \text{\text{\theta}_{j}} \text{'} (\text{dom } \text{\theta}_{j} - \text{read-tmps suspends}_{j}),() \\
\quad \text{D}_{j}, \text{acquired True } ?\text{take-sb} \_j O_{j}, R_{j}) \\
\quad \text{by (auto simp: Let-def)} \\
\textbf{from} \ \text{valid-last-prog [OF } j \text{-bound ts}_{sb-j} \) \text{have} \ \text{last-prog} : \text{last-prog } p_{j} \text{ sb}_{j} = p_{j}. \\
\textbf{from} \ \text{j-bound i-bound'} \text{ leq} \ \text{have} \ \text{j-bound-ts'} : j < \text{length ts} \\
\quad \text{by simp} \\
\quad \text{from} \ \text{read-only-read-acquired-unforwarded-acquire-witness} \ [\text{OF } nvo-drop-j \) \text{consis-drop-j} \\
\quad \text{a-not-ro-j a-unacq-take a-in]} \\
\textbf{have} \ \text{False} \\
\textbf{proof} \\
\quad \text{assume } \exists \text{sop a'} v y s z a L R W. \\
\quad \text{?drop-sb}_{j} = y s @ W r e_{sb} \text{ True } a' \text{ sop v A L R W } \neq \text{ zs } \land \ a \in A \land \\
\quad a \notin \text{outstanding-refs is-Write}_{sb} y s \land a' \neq a \\
\quad \text{with suspends}_{j} \\
\quad \text{obtain } a' \text{sop'} v' y s z s' A' L' R' W' \text{ where} \\
\end{verbatim}
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'

no-write: a \notin \text{outstanding-refs} is-Write_{sb} (ys @ [Write_{sb} 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$_{sb}$-j]
  have sharing-consis-j: sharing-consistent $S_{sb}$ O$_j$ sb$_j$.
then have A'-R': A' \cap R' = \{

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-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) 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$_j$)

hence reads-consis-ys: reads-consistent True (acquired True ?take-sb$_j$ O$_j$) 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

637
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 \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_{ab-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 \( ?\text{thesis} \)
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 \( \theta_j \) (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-ref s is-volatile-Read_{ab}
(y_s@Write_{ab} 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 (y_s @ [Write_{ab} True a' sop' v' A' L' R' W'])
(acquired True ?take-sb_j O_j) =
acquired True y_s (acquired True ?take-sb_j O_j) \cup A' - R'

638
by (simp add: acquired-append)

from flush-store-buffer-append [where sb=ys@\[Write_{ab} True a' sop' v' A' L' R' W\]
and sb'=zs', simplified,
OF j-bound-ts'is_j [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_{ab}-volatile-reads-consistent [OF no-vol-read], where
S=S]

obtain is_j'/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
(hd-prog p_j (Write_{ab} True a' sop' v' A' L' R' W'# zs')) (ys@[Write_{ab} True a' sop' v' A' L' R' W']),
 is_j',
\theta_j | (dom \theta_j − read-tmps zs'),
(), True, acquired True ys (acquired True ?take-sb \ O sb)
release ?take-sb (dom \ S sb) R sb))
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', ?\emptyset, ()
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 < length ?ts-ys. i\neq j \rightarrow (let (O_j) = map owned ?ts-ys!j 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. \(?\)drop-sb\( j = ys @ A L R W \)\# zs \( \land a \in A \land a \notin \) outstanding.refs is-Write\(_{sb} \) ys
  with suspends\(_{sb} \)
  obtain \( ys \) zs' \( A' L' R' W' \) where
  split-suspends\(_{sb} \): suspends\(_{sb} j = ys @ A L R W \# zs' \) (is suspends\(_{sb} j = ?\)suspends)
  and
  a-A': a \( \in A' \) and
  no-write: a \( \notin \) 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\(_j \) suspends\(_{sb} j = p\(_j \)
  apply (rule last-prog-same-append [where sb=?take-sb\(_j \)])
  apply (simp only: split-suspends [symmetric] suspends\(_{sb} \))
  apply simp
  done

  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 (rule causal-program-history-suffix [where sb=?take-sb\(_j \)] )
  apply (simp only: split-suspends [symmetric] suspends\(_{sb} \))
  apply (simp add: split-suspends [symmetric] suspends\(_{sb} \))
  done

  from valid-reads [OF j-bound ts\(_{sb} j\)]
  have reads-consis-j: reads-consistent False \( O_j m_{sbj} \) sb\(_j \).

  from reads-consis-j: reads-consistent False \( O_j m_{sbj} \) sb\(_j \).
  this]
  have reads-consis-m-j: reads-consistent True (acquired True ?take-sb\(_j \) \( O_j \) m suspends\(_{sb} \))
  by (simp add: m suspends\(_{sb} \))

  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-suspends [symmetric] suspends\(_{sb} \) 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 [symmetric] suspends\(_{sb} \))
  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@[\text{Ghost}_{sb} A' L' R' W']). valid-sop sop
  apply (simp only: write-sops-append)
  apply auto

640
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_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

from sharing-consis [OF j-bound ts_{sb-j}]
  have sharing-consis-j: sharing-consistent S_{sb} O_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)

  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_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@\text{Ghost}_{sb} A'L'R'W') = {}
by (auto simp add: outstanding-refs-append suspends_j [symmetric] 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) ∪ A' − R'
bysimp 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'/R_j' where
is_j': instrs zs' @ is_{sbj} = is_j' @ prog-instrs zs' and
steps-ys: (ts, m, S) ⇒ d^*
(ts[j]=(last-prog
(hd-prog p_j (Ghost sb A'L'R'W'# zs')) (ys@[Ghost sb A'L'R'W']),
is_j',
\emptyset_j \vdash (dom \emptyset_j − read-tmps zs'),
\emptyset_j)
\bigvee\text{outstanding-refs is-volatile-Write}_{sb} ys \neq \{\}, acquired True ys
(acquired True ?take-sb O_j) ∪ A' − R',R'_j])],
flush (ys@[Ghost sb A'L'R'W']) m, share (ys@[Ghost sb A'
L' R' W']) S)
(is (\cdot,\cdot,\cdot) ⇒ d^* (?ts-ys,?m-ys,?shared-ys))
bysimp add: acquired-append outstanding-refs-append)

from i-bound' have i-bound-ys: i < \text{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', ?,\emptyset, ()
D,
acquired True ?take-sb O_{sb},release ?take-sb (dom S_{sb}) R_{sb})
bysimp

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. i \neq j → (let (O_j) = map owned ?ts-ys!j in a \notin O_j)

642
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
then show False
by simp
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 i s j O i j R i j D i j θ i j sb j
  assume j-bound: j < length ts sb'
  assume ts-j: ts sb' !j = (p i,j,i s,j,sb j,D i,j,O i,j,R i,j)
  show reads-consistent False O i m sb' sb j
proof (cases i=j)
  case True
  from reads-consistent 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 < length ts sb'
  by (simp add: ts sb')
  moreover from ts-j False have ts-j': ts sb' !j = (p i,j,i s,j,sb j,D i,j,O i,j,R i,j)
  using j-bound by (simp add: ts sb')
  ultimately have consis-m: reads-consistent False O i 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 /∈ O i \ all-acquired sb j
  by simp

  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 a-unowned acquired-reads-all-acquired [of True ?take-sb j O j]
  acquired-append [of ?take-sb j ?drop-sb j]
  have a-not-acq-reads: a /∈ acquired-reads True ?take-sb j O j
  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

643
\(A=\text{unforwarded-non-volatile-reads} \ ?\text{drop-sb} \ \{\} \cup \text{acquired-reads} \ True \ ?\text{take-sb} \ O_J \ \text{and} \ m'=(m_{sb}(a:=v)), \ OF - - - \ \text{consis-m} \)

by (fastforce simp add: m_{sb}')

qed

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

fix j is_j O_j \(D_j \ \bar{v}_j \ sb_j \ p_j\)

assume j-bound: \(j < \text{length } ts_{sb}'\)

assume ts_{sb}'-j: \(ts_{sb}'\{j = (p_j, is_j, \bar{v}_j, sb_j, D_j, O_j, R_j)\}\)

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

\((S_{sb} \oplus W R \ominus A L) (O_{sb} \cup A - R) sb'\)

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': \(j < \text{length } ts_{sb}\)

by (auto simp add: ts_{sb}')

with ts_{sb}'-j False i-bound

have ts_{sb}'-j: \(ts_{sb}'\{j = (p_j, is_j, \bar{v}_j, sb_j, D_j, O_j, R_j)\}\)

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 read-only-reads-unowned [OF j-bound' i-bound False ts_{sb}' ts_{sb}-i] L-subset

have \(\forall a \in \text{read-only-reads}\)

(acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) \ sb_j) O_j)

(dropWhile (Not \circ \text{is-volatile-Write}_{sb}) \ sb_j).

a \in \text{read-only } S_{sb} \rightarrow a \in \text{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-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

next

644
show outstanding-volatile-writes-unowned-by-others ts sub

proof (unfold-locales)
  fix i_1 p_1 i_1 \ R_1 D_1 x_1 s_1 b_1 p_j i_j \ R_j D_j x_j s_j b_j
  assume i-bound: i_1 < length ts sub'
  assume j-bound: j < length ts sub'
  assume i_1-j: i_1 \neq j
  assume ts-i_1: ts sub!i_1 = (p_1,i_1,x_1,s_1,b_1,D_1,O_1,R_1)
  assume ts-j: ts sub!j = (p_j,i_j,x_j,s_j,b_j,D_j,O_j,R_j)
  show (O_j \cup all-acquired sb_j) \cap outstanding-refs is-volatile-Write sb sb = {} 
  proof (cases i_1=i)
    case True
    from i_1-j True have neq-i-j: i_1 \neq j 
    by auto
    from j-bound have j-bound': j < length ts sub
    by (simp add: ts sub'
    from ts-j neq-i-j have ts-j': ts sub!j = (p_j,i_j,x_j,s_j,b_j,D_j,O_j,R_j)
    by (simp add: ts sub'
    from outstanding-volatile-writes-unowned-by-others [OF i-bound j-bound' neq-i-j ts sub-i ts-j]
    ts-i_1 False have ts-i_1': ts sub!i_1 = (p_1,i_1,x_1,s_1,b_1,D_1,O_1,R_1)
    by (simp add: ts sub'
    show ?thesis
    proof (cases j=i)
      case True
      from outstanding-volatile-writes-unowned-by-others [OF i-bound' j-bound i_1-i ts-i_1' ts sub-i]
      have (O sb \cup all-acquired sb) \cap outstanding-refs is-volatile-Write sb sb = {}.
      then show ?thesis
      using True i_1-i ts sub-i i-bound
      by (auto simp add: sb Write sb volatile ts sub' O sb')
    next
    case False
    from j-bound have j-bound': j < length ts sub
    by (simp add: ts sub'
    from ts-j False have ts-j': ts sub!j = (p_j,i_j,x_j,s_j,b_j,D_j,O_j,R_j)
    by (simp add: ts sub'
    from outstanding-volatile-writes-unowned-by-others [OF i-bound' j-bound' i_1-j ts-i_1' ts-j']
    show (O_j \cup all-acquired sb_j) \cap outstanding-refs is-volatile-Write sb sb = {}.
    qed
    qed
    qed
next

\textbf{show} \textit{read-only-reads-unowned} ts_{sb}^{'}

\textbf{proof}

\textbf{fix} \ n \ m
\textbf{fix} \ p_{n} \ is_{n} \ \mathcal{O}_{n} \ \mathcal{R}_{n} \ \mathcal{D}_{n} \ varnothing_{n} \ sb_{n} \ p_{m} \ is_{m} \ \mathcal{O}_{m} \ \mathcal{R}_{m} \ \mathcal{D}_{m} \ varnothing_{m} \ sb_{m}

\textbf{assume} \ n \text{-bound:} \ n < \text{length ts}_{sb}^{'}
\textbf{and} \ m \text{-bound:} \ m < \text{length ts}_{sb}^{'}
\textbf{and} \ \text{neq-n-m:} \ n \neq m
\textbf{and} \ \text{nth:} \ ts_{sb}^{'}_{n} = (p_{n}, is_{n}, \varnothing_{n}, sb_{n}, \mathcal{D}_{n}, \mathcal{O}_{n}, \mathcal{R}_{n})
\textbf{and} \ \text{mth:} \ ts_{sb}^{'}_{m} = (p_{m}, is_{m}, \varnothing_{m}, sb_{m}, \mathcal{D}_{m}, \mathcal{O}_{m}, \mathcal{R}_{m})

\textbf{from} \ n \text{-bound} \ \textbf{have} \ n \text{-bound'}: \ n < \text{length ts}_{sb}^{'} \ \textbf{by} \ (\text{simp add: ts}_{sb}^{'}_{n})
\textbf{from} \ m \text{-bound} \ \textbf{have} \ m \text{-bound'}: \ m < \text{length ts}_{sb}^{'} \ \textbf{by} \ (\text{simp add: ts}_{sb}^{'}_{m})

\textbf{show} \ \((\mathcal{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}) \mathcal{O}_{n})}) \ (\text{dropWhile (Not \circ \text{is-volatile-Write}_{sb}^{'} sb_{n}) = }) = \{\}

\textbf{proof} \ (\text{cases m=i})
\textbf{case} \ True
\textbf{with} \ \text{neq-n-m} \ \textbf{have} \ \text{neq-n-i:} \ n \neq i
\textbf{by} \ \text{auto}

\textbf{with} \ n \text{-bound} \ \text{nth i-bound} \ \textbf{have} \ \text{nth'}: \ ts_{sb}^{'}_{n} = (p_{n}, is_{n}, \varnothing_{n}, sb_{n}, \mathcal{D}_{n}, \mathcal{O}_{n}, \mathcal{R}_{n})
\textbf{by} \ (\text{auto simp add: ts}_{sb}^{'}_{n})
\textbf{note} \ \text{read-only-reads-unowned} \ [\text{OF n-bound'} \ i-bound \ \text{neq-n-i nth'} \ ts_{sb}^{'}_{i}]
\textbf{then}
\textbf{show} \ ?\text{thesis}
\textbf{using} \ True \ ts_{sb}^{'}_{i} \ \text{neq-n-i nth mth n-bound'} \ m-bound' \ \text{L-subset}
\textbf{by} \ (\text{auto simp add: ts}_{sb}^{'}_{m} \mathcal{O}_{sb}^{'} \ sb_{Write_{sb}^{'} \text{volatile}})

\textbf{next}
\textbf{case} \ False
\textbf{note} \ \text{neq-m-i = this}
\textbf{with} \ m \text{-bound} \ mth i-bound \ \textbf{have} \ mth': \ ts_{sb}^{'}_{m} = (p_{m}, is_{m}, \varnothing_{m}, sb_{m}, \mathcal{D}_{m}, \mathcal{O}_{m}, \mathcal{R}_{m})
\textbf{by} \ (\text{auto simp add: ts}_{sb}^{'}_{m})
\textbf{show} \ ?\text{thesis}
\textbf{proof} \ (\text{cases n=i})
\textbf{case} \ True
\textbf{from} \ \text{read-only-reads-append} \ [\text{of (} \mathcal{O}_{sb}^{'} \cup \ A - R) \ \text{(takeWhile (Not \circ \text{is-volatile-Write}_{sb}^{'} sb_{n}) sb_{n})}]
\textbf{have} \ \text{read-only-reads}
\textbf{(acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}^{'} sb_{n}) \mathcal{O}_{n})}) \ (\text{dropWhile (Not \circ \text{is-volatile-Write}_{sb}^{'} sb_{n}) = }) \subseteq \text{read-only-reads} \ (\mathcal{O}_{sb}^{'} \cup \ A - R) \ sb_{n}
\textbf{by} \ \text{auto}

\textbf{with} \ ts_{sb}^{'}_{i} \ \text{nth mth neq-m-i n-bound'} \ True
\textbf{read-only-reads-unowned} \ [\text{OF i-bound m-bound'} \ \text{False} \ \text{symmetric} \ ts_{sb}^{'}_{i} \ mth']
\textbf{show} \ ?\text{thesis}
\textbf{by} \ (\text{auto simp add: ts}_{sb}^{'}_{i} \ sb \mathcal{O}_{sb}^{'} \ sb_{Write_{sb}^{'} \text{volatile}})
\textbf{next}

646
case False
with n-bound nth i-bound have nth': ts_{sb}'n = (p_n, is_n, θ_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

next
show ownership-distinct ts_{sb}'
proof (unfold-locales)
  fix i_1 j_1 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 ≠ j
  assume ts-i1: ts_{sb}'i_1 = (p_1, is_1, xs_1, sb_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 ∪ all-acquired sb_1) ∩ (O_j ∪ all-acquired sb_j) = {}
  proof (cases i_1 = i)
    case True
    with i_1-j have i-j: i ≠ j
    by simp
    from ownership-distinct [OF i-bound j-bound' j-bound' i-bound] ts_{sb}'i
    show ?thesis
      using ts_{sb}'i True ts-j i-bound O_{sb}'
      by (auto simp add: ts_{sb}' sb Write_{sb} volatile)
  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}')
    hence i_1-bound'': i_1 < 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 ownership-distinct [OF i-bound' j-bound' i-j ts_{sb}' i-j ts-j']
    show ?thesis
    using ts_{sb}'i True ts-i_1 i-bound O_{sb}'
    by (auto simp add: ts_{sb}' sb Write_{sb} volatile)
  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}')
    hence i_1-bound'': i_1 < length (map owned ts_{sb})
    by simp
    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 ownership-distinct [OF i-bound' j-bound' i-i ts-i_1' ts_{sb}' i]
        show ?thesis
        using ts_{sb}'i True ts-j i-bound O_{sb}'
by (auto simp add: ts_{sb}′ sb \textit{Write}_{sb} \text{ volatile})

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 ownership-distinct [OF i-bound ts-i′ i-bound jth ts-j′]
show ?thesis .

qed

have valid-sharing′: valid-sharing (S_{sb} ⊕ W R ⊕ A L) ts_{sb}′
proof (intro-locales)
  show outstanding-non-volatile-writes-unshared (S_{sb} ⊕ W R ⊕ 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-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 non-volatile-writes-unshared (S_{sb} ⊕ W R ⊕ A L) sb_j
    proof (cases i=j)
      case True
      with outstanding-non-volatile-writes-unshared [OF i-bound ts-i′ i-bound jth ts-j′]
      i-bound jth ts_{sb}′ i show ?thesis
      by (clarsimp simp add: ts_{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, 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′ jth′]
      have unshared: non-volatile-writes-unshared S_{sb} sb_j.

      have ∀a∈\text{dom} (S_{sb} ⊕ W R ⊕ A L) − \text{dom} S_{sb}, a \notin \text{outstanding-refs}
      is-non-volatile-Write_{sb} sb_j
      proof −
      {
        fix a
        assume a-in: a ∈ \text{dom} (S_{sb} ⊕ W R ⊕ A L) − \text{dom} S_{sb}
        hence a-R: a ∈ R
          by clarsimp
        assume a-in-j: a ∈ \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

  648
have a ∈ O_j ∪ all-acquired sb_j
by auto

moreover
with ownership-distinct [OF i-bound j-bound' False ts_{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_{sb} 
[OF unshared this]
show ?thesis .
qed
qed
next
show sharing-consis (S_{sb} ⊕ W R ⊃ A L) ts_{sb'}
proof (unfold-locales)
fix j p_j is_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, R_j, O_j)
show sharing-consistent (S_{sb} ⊕ W R ⊃ A L) O_j sb_j
proof (cases i=j)
case True
with i-bound jth ts_{sb}-i sharing-consis [OF i-bound ts_{sb}-i]
show ?thesis
by (clarsimp simp add: ts_{sb'} sb Write_{sb} volatile O_{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, is_j, xs_j, sb_j, D_j, O_j, R_j)
by (auto simp add: ts_{sb}')
from sharing-consis [OF j-bound' jth']
have consis: sharing-consistent S_{sb} O_j sb_j.

have acq-cond: all-acquired sb_j ∩ dom S_{sb} − dom (S_{sb} ⊕ W R ⊃ A L) = {}
proof –
{
fix a
assume a-acq: a ∈ all-acquired sb_j
assume a ∈ dom S_{sb}
assume a-L: a ∈ L
have False
proof –
from ownership-distinct [OF i-bound j-bound' False ts_{sb}-i jth']
have \( A \cap \text{all-acquired } S_b_j = \{ \} \)
by (auto simp add: sb_WRITE sb_volatile)
with \( \text{a-}\text{acq a-L L-subset} \)
show False
by blast
qed
}
thus ?thesis
by auto
qed

have uns-cond: \( \text{all-unshared } S_b_j \cap \text{dom } (S_{sb} \oplus_W R \ominus_A L) - \text{dom } S_{sb} = \{ \} \)
proof -
{
fix a
assume a-uns: \( a \in \text{all-unshared } S_b_j \)
assume a \( \notin L \)
assume a-R: \( a \in R \)
have False
proof -
from unshared-acquired-or-owned [OF consis] a-uns
have \( a \in \text{all-acquired } S_b_j \cup O_j \) by auto
with ownership-distinct [OF i-bound j-bound \( \text{False ts}_{sb-i} \text{ jth} \) R-owned a-R
show False
by blast
qed
}
thus ?thesis
by auto
qed

next
show read-only-unowned \( (S_{sb} \oplus_W R \ominus_A L) \text{ } ts_{sb} \)'
proof
fix \( j \) \( p_j \) is_j \( O_j \) \( R_j \) \( D_j \) \( x_{s_j} \) \( S_{sb} \)
assume j-bound: \( j < \text{length } ts_{sb} \)'
assume jth: \( ts_{sb} \) ! \( j = (p_j, is_j, x_{s_j}, S_{sb}, D_j, O_j, R_j) \)
show \( O_j \cap \text{read-only } (S_{sb} \oplus_W R \ominus_A L) = \{ \} \)
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: \( O_{sb} \text{ } ts_{sb} \)')

650
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 read-only-unowned [OF j-bound' jth'] have O_j ∩ read-only S_sb = {}.
  moreover
  from ownership-distinct [OF i-bound j-bound' False ts'_sb-i jth'] R-owned
  have (O_sb ∪ A) ∩ 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 ⊕ W R ⊖ A L) ts'_sb
proof (unfold-locales)
  show − ∪ ((λ(−, −, −, −, O, r), O) i set ts'_sb) ⊆ dom (S_sb ⊕ W R ⊖ A L)
  proof
  have s: ∪ ((λ(−, −, −, −, O, r), O) i set ts'_sb) =
  ∪ ((λ(−, −, −, −, O, r), O) i 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 no-outstanding-write-to-read-only-memory (S_sb ⊕ W R ⊖ A L) ts'_sb
proof
  fix j p_j is_j O_j R_j D_j acq_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 ⊕ W R ⊖ 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\sb 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′)
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 ∩ 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 sb_j = {}
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 sb_j = {}
by (auto simp add: misc-outstanding-refs-convs)
with R-owned
show ?thesis by blast
qed
then
have ∀ a ∈ outstanding-refs is-Write\sb sb_j.
a ∈ read-only (S\sb ⊕ \ WR ⊖ \ AL) ⇒ 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
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) →
(is′, ?\theta , (), m (a := v),True, acquired True ?take-sb O\sb ∪ A − R, Map.empty,S ⊕\WR R ⊖\AL)
by (simp add: f-v′ [symmetric])

from direct-computation.Memop [OF i-bound′ ts-i this]
have store-step:
(ts, m, S) ⇒\sb (ts′, m(a := v),S ⊕\WR R ⊖\AL).

have sb′-split:
\[ \begin{align*}
\text{sb}' &= \text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}' \@ \\
&\quad \text{dropWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}' \\
&\quad \text{by simp}
\end{align*} \]

\textbf{from reads-consis}

\textbf{have no-vol-reads: outstanding-refs is-volatile-Read}_{sb} \text{ sb}' = \{\}
\quad \text{by (simp add: sb Write}_{sb} \text{ True)}

\textbf{hence outstanding-refs is-volatile-Read}_{sb} (\text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}')
\quad = \{\}
\quad \text{by (auto simp add: outstanding-refs-conv dest: set-takeWhileD)}

\textbf{moreover}

\textbf{have outstanding-refs is-volatile-Write}_{sb}
\quad (\text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}') = \{\}

\textbf{proof —}

\textbf{have \( \forall r \in \text{ set } \) (\text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}'). \neg (\text{is-volatile-Write}_{sb} r)}
\quad \text{by (auto dest: set-takeWhileD)}

\textbf{thus \( ?\text{thesis} \)}
\quad \text{by (simp add: outstanding-refs-conv)}

\textbf{qed}

\textbf{ultimately}

\textbf{have no-volatile:}
\quad outstanding-refs is-volatile (\text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}') = \{\}
\quad \text{by (auto simp add: outstanding-refs-conv is-volatile-split)}

\textbf{moreover}

\textbf{from no-vol-reads have \( \forall r \in \text{ set sb}' \) . \neg \text{is-volatile-Read}_{sb} r}
\quad \text{by (fastforce simp add: outstanding-refs-conv is-volatile-Read}_{sb}\text{-def split: memref.splits)}

\textbf{hence \( \forall r \in \text{ set sb}' . \) (\text{Not } \not \circ \text{is-volatile-Write}_{sb}) r = (\text{Not } \not \circ \text{is-volatile}) r}
\quad \text{by (auto simp add: is-volatile-split)}

\textbf{hence takeWhile-eq: (takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}')
\quad = (\text{takeWhile } (\not \circ \text{is-volatile}) \text{ sb}']
\quad \text{apply —}
\quad \text{apply (rule takeWhile-cong)}
\quad \text{apply auto}
\quad \text{done}

\textbf{from leq}

\textbf{have leq'}: length ts_{sb} = length ?ts'
\quad \text{by simp}

\textbf{hence i-bound-ts'}: i < length ?ts' \textbf{using i-bound by simp}

\textbf{from is'}-sim

\textbf{have is'}-sim-split:
\quad instrs
\quad \quad (\text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}' \@ \\
\quad &\quad \text{dropWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}' \@ \text{is}_{sb} = \\
\quad &\quad \text{is'} \@ \text{prog-instrs } (\text{takeWhile } (\not \circ \text{is-volatile-Write}_{sb}) \text{ sb}' \@
dropWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′)
by (simp add: sb′-split [symmetric])

from reads-consistent-flush-all-until-volatile-write [OF valid-ownership-and-sharing \textsubscript{S\textsubscript{sb}}
\textsubscript{ts\textsubscript{sb}}]
i-bound ts\textsubscript{sb}-i reads-consis]
have reads-consistent True (acquired True ?take-sb \textsubscript{O\textsubscript{sb}} m (Write\textsubscript{sb} True a (D,f) v A L R W\#sb′)
by (simp add: m sb Write\textsubscript{sb} volatile)
hence reads-consistent True (acquired True ?take-sb \textsubscript{O\textsubscript{sb}} ∪ A − R) (m(a:=v)) sb′
by simp
from reads-consistent-takeWhile [OF this]
have r-consis′; reads-consistent True (acquired True ?take-sb \textsubscript{O\textsubscript{sb}} ∪ A − R) (m(a:=v))
(takeWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′).

from last-prog have last-prog-sb′; last-prog p\textsubscript{sb} sb′ = p\textsubscript{sb}
by (simp add: sb Write\textsubscript{sb})

from valid-write-sops [OF i-bound ts\textsubscript{sb}-i]
have ∀sop ∈ write-sops sb′. valid-sop sop
by (auto simp add: sb Write\textsubscript{sb})
hence valid-sop′; ∀sop∈write-sops (takeWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′).
valid-sop sop
by (fastforce dest: set-takeWhileD simp add: in-write-sops-conv)

from no-volatile
have no-volatile-Read\textsubscript{sb}
outstanding-refs is-volatile-Read\textsubscript{sb} (takeWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′) =
\{\}
by (auto simp add: outstanding-refs-conv is-volatile-Read\textsubscript{sb}-def split: memref.splits)
from flush-store-buffer-append [OF i-bound-ts′ is′-sim-split, simplified,
OF causal-program-history-sb′ ts′-i refl last-prog-sb′ r-consis′ hist-consis′
valid-sop′ dist-sb′ no-volatile-Read\textsubscript{sb}-volatile-reads-consistent [OF no-volatile-Read\textsubscript{sb}],
where \textbf{S}=(S ⊕_W R ⊕_A L)]

obtain is′′ where
is′′-sim: instrs (dropWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′) @ is\textsubscript{sb} =
is′′ @ prog-instrs (dropWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′) and
steps: (?ts′, m(a := v), S ⊕_W R ⊕_A L) ⇒\textsubscript{d} s∗
(ts[i := (last-prog (hd-prog p\textsubscript{sb} (dropWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′))))
(takeWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′),
is′′,
\textbf{v}_{\textsubscript{sb}} | (dom \textbf{v}_{\textsubscript{sb}} −
read-tmps (dropWhile (Not ◦ is-volatile-Write\textsubscript{sb}) sb′)),

654
(\), True, acquired True (takeWhile (Not o is-volatile-Write) sb) sb'
(acquired True ?take-sb O sb ∪ A − R),
release (takeWhile (Not o is-volatile-Write) sb) sb'
(dom (S ⊕ W R ⊕ A L) Map.empty),
flush (takeWhile (Not is-volatile-Write) sb sb') (m(a := v)),
share (takeWhile (Not 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-volatile-commute [OF i-bound ts sb-i [simplified sb Write sb True]
outstanding-refs-is-Write-takeWhile-disj a-notin-others']

from last-prog-hd-prog-append' [where sb=(takeWhile (Not is-volatile-Write) sb) sb']
and sb'==(dropWhile (Not 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 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) (r#sb) = []
by (simp add: Write sb True)

have dist-sb': \forall p is O R D \not\in sb.
i < length ts sb \rightarrow
ts sb ! i = (p, is, \theta, sb, D, O, R) \rightarrow
(all-shared (takeWhile (Not o is-volatile-Write) sb) sb) \cup
all-unshared (takeWhile (Not o is-volatile-Write) sb sb) \cup
all-acquired (takeWhile (Not o is-volatile-Write) sb sb') \cap
(all-shared (takeWhile (Not o is-volatile-Write) sb sb') sb) \cup
all-unshared (takeWhile (Not o is-volatile-Write) sb sb) sb') \cup
all-acquired (takeWhile (Not o is-volatile-Write) sb sb') sb') =
{}

proof

{ 
fix j p j is j O j R j D j \theta j sb j x
assume j-bound: j < length ts sb
assume jth: ts sb\{j = (p j is j \theta j sb j D j O j R j)
assume x-shared: x \in all-shared (takeWhile (Not o is-volatile-Write) sb sb) sb)
all-unshared (takeWhile (Not o is-volatile-Write) sb sb) sb]
all-acquired (takeWhile (Not o is-volatile-Write) sb sb) sb]
assume x-sb: x \in (all-shared (takeWhile (Not o is-volatile-Write) sb sb') sb sb') \cup
all-unshared (takeWhile (Not o is-volatile-Write) sb sb') sb]

655
all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb')

have False

proof (cases i=j)

case True with x-shared ts_{sb'-i} jth show False by (simp add: sb volatile Write_{sb})

next

case False

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)

all-acquired-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb_j)]

have x ∈ all-acquired sb_j ∪ 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 (Not ◦ is-volatile-Write_{sb}) sb')]

(dropWhile (Not ◦ is-volatile-Write_{sb}) sb')

all-unshared-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb')]

(dropWhile (Not ◦ is-volatile-Write_{sb}) sb')

all-acquired-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb')]

(dropWhile (Not ◦ is-volatile-Write_{sb}) sb')

have x ∈ all-acquired sb ∪ O_{sb}

by (auto simp add: sb Write_{sb} volatile)

moreover

note ownership-distinct [OF i-bound j-bound False ts_{sb'-i} jth]

ultimately show False by blast

qed

}

thus ?thesis by blast

qed

have dist-R-L-A: ∀ j p is O R D θ sb.

j < length ts_{sb} → i ≠ j →

ts_{sb} ! j = (p, is, θ, sb, D, O, R) →

(all-shared sb ∪ all-unshared sb ∪ all-acquired sb) ∩ (R ∪ L ∪ A) = {}

proof –

{ fix j p_j θ_j D_j O_j R_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, θ_j, sb_j, D_j, O_j, R_j)

assume x-shared: x ∈ all-shared sb_j ∪

all-unshared sb_j ∪

all-acquired sb_j

assume x-R-L-A: x ∈ R ∪ L ∪ A

656
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]]

have x ∈ all-acquired sb j ∪ O j
by auto
moreover
from x-R-L-A R-owned L-subset
have x ∈ all-acquired sb ∪ O sb
by (auto simp add: sb Write sb volatile)
moreover
note ownership-distinct [OF i-bound j-bound neq-i-j ts sb-i jth]
ultimately show False by blast
qed

thus ?thesis by blast
qed
from local.ownership-distinct-axioms have ownership-distinct ts sb .
from local.sharing-consis-axioms have sharing-consis S sb ts sb .

note share-commute=
share-all-until-volatile-write-flush-commute [OF take-empty (ownership-distinct ts sb) (sharing-consis S sb ts sb) i-bound ts sb-i dist sb dist-R-L-A]

have rel-commute-empty:
release (takeWhile (Not ◦ is-volatile-Write sb) sb') (dom S ∪ R − L) Map.empty =
release (takeWhile (Not ◦ is-volatile-Write sb) sb') (dom S sb ∪ R − L)
Map.empty
proof -
{
fix a
assume a-in: a ∈ all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb')
have (a ∈ (dom S ∪ R − L)) = (a ∈ (dom S sb ∪ R − L))
proof -

from all-shared-acquired-or-owned [OF sharing-consis [OF i-bound ts sb-i]] a-in
all-shared-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb') (dropWhile (Not ◦ is-volatile-Write sb) sb')]
have a ∈ O sb ∪ all-acquired sb
by (auto simp add: sb Write sb volatile)
from share-all-until-volatile-write-thread-local [OF (ownership-distinct ts sb) (sharing-consis S sb ts sb) i-bound ts sb-i this]
have S a = S sb a
by (auto simp add: sb Write sb volatile S)
then show ?thesis
by (auto simp add: domIff)
qed
}
then show ?thesis
apply –
apply (rule release-all-shared-exchange)
apply auto
done
qed

{ 
fix \( j \) \( p_j \) \( is_j \) \( O_j \) \( D_j \) \( v_j \) \( sb_j \) \( x \)
assume \( j \text{th: } ts_{sb}^j = (p_j, is_j, v_j, sb_j, D_j, O_j, R_j) \)
assume \( j \text{-bound: } j < \text{length } ts_{sb}^j \)
assume neq: \( i \neq j \)
have release \( (\text{takeWhile } (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \)
\( (\text{dom } S_{sb} \cup R - L) \text{ R}_j \)
= release \( (\text{takeWhile } (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \)
\( (\text{dom } S_{sb}) \text{ R}_j \)

proof –
{}
fix a
assume a-in: \( a \in \text{all-shared } (\text{takeWhile } (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \)
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-dist: \( A \cap (O_j \cup \text{all-acquired } sb_j) = \{\} \)
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 \text{is-volatile-Write}_{sb}) \text{ sb}_j) ]
\( (\text{dropWhile } (Not \circ \text{is-volatile-Write}_{sb}) \text{ sb}_j) \)

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 \( a \notin (O_{sb} \cup \text{all-acquired } sb) \) 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
apply –
apply (rule release-all-shared-exchange)
apply auto
done
qed

note release-commute = this
have \( (\text{ts}_{\text{sb}} [i := (p_{\text{sb}}, \text{is}_{\text{sb}}, \text{ts}_{\text{sb}}), D_{\text{sb}}, O_{\text{sb}} \cup A - R, \text{Map}.empty]) , m_{\text{sb}}(a := v), S_{\text{sb}}') \sim \\
(\text{ts}[i := (\text{last-prog} (\text{hd-prog} p_{\text{sb}} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}')))) \\
\quad \text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}'), \\
\quad \text{is}'', \\
\quad \text{ts}_{\text{sb}} |' (\text{dom} \text{ts}_{\text{sb}} - \\
\quad \text{read-tmps} (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}'))), \\
\quad (True, \text{acquired} True (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}')) \\
\quad (\text{acquired} True \ ?\text{take-sb} O_{\text{sb}} \cup A - R), \\
\quad \text{release} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}')) \\
\quad (\text{dom} (S \oplus W R \ominus A L)) \text{Map}.empty), \\
\quad \text{flush} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}')) (m(a := v)), \\
\quad \text{share} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{\text{sb}}) s_{\text{sb}}')) (S \oplus W R \ominus A L)) \\
\text{apply} \ (\text{rule} \ \text{sim-config}.\text{intros}) \\
\text{apply} \ (\text{simp} \ \text{add}: \text{flush-commute} \ m) \\
\text{apply} \ (\text{clarsimp} \ \text{simp} \ \text{add}: S_{\text{sb}}' \ S \text{share-commute} \ \text{simp} \ \text{del}: \text{restrict-restrict}) \\
\text{using} \ \text{leq} \\
\text{apply} \ \text{clarsimp} \\
\text{using} \ i\text{-bound} \ i\text{-bound}' \text{ts-sim} \ D \\
\text{apply} \ (\text{clarsimp} \ \text{simp} \ \text{add}: \text{Let-def} \ \text{nth-list-update} \ \text{is}'-\text{sim} \ \text{last-prog-eq} \ \text{sb} \ \text{Write}_{\text{sb}} \ \text{volatile} \ S_{\text{sb}}') \\
\quad \text{rel-commute-empty} \\
\quad \text{split: if-split-asm} \ ) \\
\quad \text{apply} \ (\text{rule} \ \text{conjI}) \\
\quad \text{apply} \ \text{blast} \\
\quad \text{apply} \ \text{clarsimp} \\
\quad \text{apply} \ (\text{frule} \ (2) \ \text{release-commute}) \\
\quad \text{apply} \ \text{clarsimp} \\
\quad \text{apply} \ \text{fastforce} \\
\text{done} \\

\text{ultimately} \\
\text{show} \ \?\text{thesis} \\
\text{using} \ \text{valid-own} \ \text{valid-hist} \ \text{valid-reads} \ \text{valid-sharing} \ \text{tmps-distinct} \ \\
\text{valid-dd} \ \text{valid-sops} \ \text{load-tmps-fresh} \ \text{enough-flushs} \\
\text{valid-program-history} \ \text{valid} \\
\text{m}_{\text{sb}}', S_{\text{sb}}', \text{ts}_{\text{sb}}' \\
\text{by} \ (\text{auto} \ \text{simp} \ \text{del}: \text{fun-upd-apply} \ \text{simp} \ \text{add}: O_{\text{sb}}', R_{\text{sb}}', \text{O}_{\text{sb}}') \\

\text{next} \\
\text{case} \ False \\
\text{note} \ \text{non-vol} = \text{this} \\

\text{from} \ \text{flush} \ \text{Write}_{\text{sb}} \ False \\
\text{obtain} \\
\quad O_{\text{sb}}': O_{\text{sb}}'=O_{\text{sb}} \ \text{and} \\
\quad S_{\text{sb}}': S_{\text{sb}}'=S_{\text{sb}} \ \text{and} \\
\quad R_{\text{sb}}': R_{\text{sb}}'= R_{\text{sb}} \\
\text{by} \ \text{cases} \ (\text{auto} \ \text{simp} \ \text{add}: \text{sb})
from non-volatile-owned non-vol have a-owned: a ∈ \( \mathcal{O}_{sb} \)

by simp

\{
  
  fix \( j \)
  fix \( p_j \is_{sbj} \mathcal{O}_j \ D_{sbj} \emptyset \mathcal{R}_j \sb_j \)
  assume j-bound: \( j < \text{length } ts_{sb} \)
  assume ts_{sb}-j: ts_{sb}'\|=(p_j,\is_{sbj};\emptyset;sb_j,D_{sbj}.;\mathcal{O}_j.,\mathcal{R}_j)
  assume neq-i-j: \( i \neq j \)
  have a /∈ unforwarded-non-volatile-reads (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb_j) \{ \}

proof

let \(?\text{take-sb}_j = \text{takeWhile (Not } \circ \text{is-volatile-Write}_{sb}) \sb_j \)
let \(?\text{drop-sb}_j = \text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb}) \sb_j \)
assume a-in: a ∈ 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 /∈ \mathcal{O}_j and a-unacq: a /∈ all-acquired sb_j

by auto

with all-acquired-append [of ?take-sb_j ?drop-sb_j]
acquired-takeWhile-non-volatile-Write_{sb} [of sb_j \mathcal{O}_j]
have a-unacq-take: a /∈ acquired True ?take-sb_j \mathcal{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-drop (share ?take-sb_j \mathcal{S}_{sb})
  (acquired True ?take-sb_j \mathcal{O}_j) ?drop-sb_j .

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,

from non-volatile-owned-or-read-only-outstanding-refs [OF nvo-drop-j] a-in'
have a ∈ acquired True ?take-sb_j \mathcal{O}_j ∪ all-acquired ?drop-sb_j ∪
  read-only-reads (acquired True ?take-sb_j \mathcal{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 \mathcal{O}_j] acquired-all-acquired [of True ?take-sb_j \mathcal{O}_j]
  all-acquired-append [of ?take-sb_j ?drop-sb_j]

have acquired True ?take-sb_j \mathcal{O}_j ∪ all-acquired ?drop-sb_j ⊆ \mathcal{O}_j ∪ all-acquired sb_j
by auto

ultimately

have a ∈ read-only-reads (acquired True ?take-sb_j \mathcal{O}_j) ?drop-sb_j
using a-owned ownership-distinct [OF i-bound j-bound neq-i-j ts_{sb}-i ts_{sb}-j]
by auto
have valid-reads\': valid-reads m_{sb}' ts_{sb}'

proof (unfold-locales)
  fix j p j \is\ j \O\ j D j \ubar j sb j
  assume j-bound: j < length ts_{sb}'
  assume ts-j: ts_{sb} ! j = (p j, \is j, \ubar j, sb j, D j, \O j, R j)
  show \text{reads-consistent False } \O j m_{sb}' sb j
    proof (cases i=j)
      case True
      from \text{reads-consis ts-j j-bound sb show } ?\text{thesis}
        by (clarsimp simp add: True m_{sb} ts_{sb} O_{sb} ts_{sb}' O_{sb}' False
          reads-consistent-pending-write-antimono)
      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, \is j, \ubar j, sb j, D j, \O j, R j)
        using j-bound by (simp add: ts_{sb}')
      ultimately have \text{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 all-acquired sb j
          by simp
        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
        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 acquired-reads True ?take-sb j \O j
            by auto
          moreover
        note a-unfw= a-notin-unforwarded-non-volatile-reads-drop [OF j-bound' ts-j' False]
        ultimately
        show ?\text{thesis}
          using \text{reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop} [where
            W=\{\} and
            A=unforwarded-non-volatile-reads ?drop-sb j \{\} \cup acquired-reads True ?take-sb j \O j and
            m'= (m_{sb}(a:=v)), OF - - - consis-m]
            by (fastforce simp add: m_{sb}')
      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}] sb
    have non-volatile-owned-or-read-only False \( S_{sb} \) \( O_{sb} \) sb'
    by (auto simp add: Write_{sb} False)
    from outstanding-non-volatile-refs-owned-or-read-only-nth-update [OF i-bound this]
    show ?thesis by (simp add: ts_{sb'} Write_{sb} False \( O_{sb'} \) \( S_{sb'} \))
  qed

next
  show outstanding-volatile-writes-unowned-by-others ts_{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: Write_{sb} False)
    have acq: all-acquired sb' \( \subseteq \) all-acquired sb
    by (auto simp add: Write_{sb} False 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'} Write_{sb} False \( O_{sb'} \))
  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 Write_{sb} non-vol)
    have \( O_{sb} \cup \) all-acquired sb' \( \subseteq \) \( O_{sb} \cup \) all-acquired sb
    by (auto simp add: sb Write_{sb} non-vol)
    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: 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

qed
have valid-sharing'. valid-sharing $S_{sb}'$, ts_{sb}'

proof (intro-locals)
  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 ◦ is-volatile-Write sb) sb') @ is_{sb} =
  is @ prog-instrs (dropWhile (Not ◦ is-volatile-Write sb) sb')
    by (simp add: suspends sb Write sb False)

have (ts,m,$S$) ⇒ₜ* (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', sb, OF - i-bound ts_{sb}-i, simplified sb]

\text{Write}_{sb} \text{ False, simplified]}

\textbf{have} (ts_{sb} [i := (p_{sb}, is_{sb}, o_{sb}', D_{sb}, O_{sb}, R_{sb})], m_{sb}(a:=v), S_{sb}') \sim

(ts, m, S)

\textbf{apply} (rule sim-config.intros)

\textbf{apply} (simp add: m flush-commute)

\textbf{apply} (clarsimp simp add: S S_{sb}' share-commute)

\textbf{using} leq

\textbf{apply} simp

\textbf{using} i-bound i-bound' is-sim ts-i ts-sim D

\textbf{apply} (clarsimp simp add: Let-def nth-list-update suspends sb Write sb False S sb' split: if-split-asm )

\textbf{done}

\textbf{ultimately}

\textbf{show} ?thesis

\textbf{using} valid-own' valid-hist' valid-reads' valid-sharing' tmps-distinct' m_{sb}'

valid-dd' valid-sops' load-tmps-fresh' enough-flushes' valid-program-history' valid'

ts_{sb}' O_{sb}' S_{sb}' R_{sb}'

\textbf{by} (auto simp del: fun-upd-apply)

\textbf{qed}

\textbf{next}

\textbf{case} (Read_{sb} volatile a t v)

\textbf{from} flush this \textbf{obtain} m_{sb}': m_{sb}'=m_{sb} \textbf{and}

O_{sb}': O_{sb}'=O_{sb} \textbf{and} S_{sb}': S_{sb}'=S_{sb} \textbf{and}

R_{sb}': R_{sb}'=R_{sb}

\textbf{by} cases (auto simp add: sb)

\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 False S_{sb} O_{sb} sb'

\textbf{by} (auto simp add: Read_{sb})

\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}' Read_{sb} O_{sb}' S_{sb}')

\textbf{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} sb

\textbf{by} (auto simp add: Read_{sb})

\textbf{have} acq: all-acquired sb' \subseteq all-acquired sb

\textbf{by} (auto simp add: Read_{sb} 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}' Read_{sb} O_{sb}')

664
qed
next
show read-only-reads-unowned \(\mathcal{O}_{\mathcal{S}b}\)′
proof –
  have ro: read-only-reads (acquired True (takeWhile (Not \(\circ\) is-volatile-Write\(\mathcal{S}b\)) \(\mathcal{O}_{\mathcal{S}b}\))
    (dropWhile (Not \(\circ\) is-volatile-Write\(\mathcal{S}b\)) \(\mathcal{O}_{\mathcal{S}b}\))
    \(\subseteq\) read-only-reads (acquired True (takeWhile (Not \(\circ\) is-volatile-Write\(\mathcal{S}b\)) \(\mathcal{S}b\))
    (dropWhile (Not \(\circ\) is-volatile-Write\(\mathcal{S}b\)) \(\mathcal{S}b\))
    by (auto simp add: \(\mathcal{S}b\) Read \(\mathcal{O}_{\mathcal{S}b}\))
  have \(\mathcal{O}_{\mathcal{S}b}\) \(\cup\) all-acquired \(\mathcal{S}b\)′ \(\subseteq\) \(\mathcal{O}_{\mathcal{S}b}\) \(\cup\) all-acquired \(\mathcal{S}b\)
    by (auto simp add: Read \(\mathcal{S}b\))
  from read-only-reads-unowned-nth-update \(\text{OF i-bound ts}_{\mathcal{S}b\text{-i}} \text{ro this}\)
  show \(?\text{thesis}\)
    by (simp add: \(\text{ts}_{\mathcal{S}b}\)′ \(\mathcal{S}b\) \(\mathcal{O}_{\mathcal{S}b}\)′)
qed

next
show ownership-distinct \(\mathcal{S}b\)′
proof –
  have acq: all-acquired \(\mathcal{S}b\)′ \(\subseteq\) all-acquired \(\mathcal{S}b\)
    by (auto simp add: Read \(\mathcal{S}b\))
  with ownership-distinct-instructions-read-value-store-buffer-independent
    \(\text{OF i-bound ts}_{\mathcal{S}b\text{-i}}\)
  show \(?\text{thesis}\) by (simp add: \(\text{ts}_{\mathcal{S}b}\)′ \(\mathcal{S}b\) \(\mathcal{O}_{\mathcal{S}b}\)′)
qed

have valid-sharing′: valid-sharing \(\mathcal{S}_{\mathcal{S}b}\)′ \(\text{ts}_{\mathcal{S}b}\)′
proof (intro-locales)
from outstanding-non-volatile-writes-unshared \(\text{OF i-bound ts}_{\mathcal{S}b\text{-i}}\)
have non-volatile-writes-unshared \(\mathcal{S}_{\mathcal{S}b}\) \(\mathcal{S}b\)′
  by (auto simp add: \(\mathcal{S}b\) Read \(\mathcal{S}b\))
from outstanding-non-volatile-writes-unshared-nth-update \(\text{OF i-bound this}\)
show outstanding-non-volatile-writes-unshared \(\mathcal{S}_{\mathcal{S}b}\)′ \(\text{ts}_{\mathcal{S}b}\)′
  by (simp add: \(\text{ts}_{\mathcal{S}b}\)′ \(\mathcal{S}_{\mathcal{S}b}\)′)
next
from sharing-consis \(\text{OF i-bound ts}_{\mathcal{S}b\text{-i}}\)
have sharing-consistent \(\mathcal{S}_{\mathcal{S}b}\) \(\mathcal{S}b\)′
  by (auto simp add: \(\mathcal{S}b\) Read \(\mathcal{S}b\))
from sharing-consis-nth-update \(\text{OF i-bound this}\)
show sharing-consis \(\mathcal{S}_{\mathcal{S}b}\)′ \(\text{ts}_{\mathcal{S}b}\)′
  by (simp add: \(\mathcal{S}_{\mathcal{S}b}\)′ \(\mathcal{O}_{\mathcal{S}b}\)′ \(\mathcal{S}_{\mathcal{S}b}\)′)
next
from read-only-unowned-nth-update \(\text{OF i-bound read-only-unowned ts}_{\mathcal{S}b\text{-i}}\)
show read-only-unowned \(\mathcal{S}_{\mathcal{S}b}\)′ \(\text{ts}_{\mathcal{S}b}\)′
  by (simp add: \(\mathcal{S}_{\mathcal{S}b}\)′ \(\mathcal{S}_{\mathcal{S}b}\)′ \(\mathcal{O}_{\mathcal{S}b}\)′)
next
from unowned-shared-nth-update \(\text{OF i-bound ts}_{\mathcal{S}b\text{-i}}\) subset-refl
show unowned-shared \(\mathcal{S}_{\mathcal{S}b}\)′ \(\text{ts}_{\mathcal{S}b}\)′
  by (simp add: \(\mathcal{S}_{\mathcal{S}b}\)′ \(\mathcal{O}_{\mathcal{S}b}\)′ \(\mathcal{S}_{\mathcal{S}b}\)′)

665
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 Read\_sb)
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
show no-outstanding-write-to-read-only-memory \(S_{sb}'/ts_{sb}'\)
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 Read\_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]
have causal-program-history is\_sb sb .
then have causal': causal-program-history is\_sb sb'
by (simp add: sb Read\_sb causal-program-history-def)

from valid-last-prog [OF i-bound ts\_sb\_i]
have last-prog \(p_{sb}/sb = p_{sb}\).
then have last-prog \(p_{sb}/sb'\ = p_{sb}
by (simp add: sb Read\_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 Read\_sb suspends)

from valid-history [OF i-bound ts\_sb\_i]
have \(θ_{sb-v}/: θ_{sb} t = Some v\)
by (simp add: history-consistent-access-last-read sb Read\_sb split:option.splits)

have \((ts,m,S) ⇒_{d}^{*} (ts,m,S)\) by blast

moreover

note flush-commute= flush-all-until-volatile-write-Read\_sb-commute [OF i-bound ts\_sb\_i
[simplified sb Read\_sb]]
note share-commute =
  share-all-until-volatile-write-update-sb [of sb' sb, OF - i-bound ts_{sb'}-i, simplified sb]
Read_{sb}, simplified]
  have (ts_{sb} [i := (p_{sb}, is_{sb}, \theta_{sb}, D_{sb}, \mathcal{O}_{sb}, \mathcal{R}_{sb}'), m_{sb}, S_{sb}']) \sim (ts, m, \mathcal{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-list' valid-reads' valid-sharing' tmms-distinct' m_{sb}'
  valid-dd' valid-sops' load-tmms-fresh' enough-flushs' valid-sharing'
valid-program-history' valid'
  ts_{sb}' \mathcal{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} and
  \mathcal{O}_{sb}': \mathcal{O}_{sb}=\mathcal{O}_{sb} and S_{sb}': S_{sb}=S_{sb} and
  \mathcal{R}_{sb}': \mathcal{R}_{sb}=\mathcal{R}_{sb}
by cases (auto simp add: sb)

  have valid-own': valid-ownershipship 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} \mathcal{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} \mathcal{O}_{sb}' S_{sb}')
  qed
next
show outstanding-volatile-writes-unowned-by-others ts_{sb}'
proof
   from sb
   have out: outstanding-refs is-volatile-\textit{Write}_{sb} sb' \subseteq outstanding-refs is-volatile-\textit{Write}_{sb} sb
   by (auto simp add: Prog_{sb})
   have acq: all-acquired sb' \subseteq 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)
  with ownership-distinct-instructions-read-value-store-buffer-independent
    [OF i-bound ts sb-i]
  show ?thesis by (simp add: ts sb' Prog sb O sb')
qed

next

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 \'Osb', \osb 'S_{sb}')

next

from no-outstanding-write-to-read-only-memory [OF i-bound ts \sb -i]
have no-write-to-read-only-memory \osb \sb' by (auto simp add: sb Prog_{sb})
from no-outstanding-write-to-read-only-memory-nth-update [OF i-bound this]
show no-outstanding-write-to-read-only-memory \osb '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 \osb m_{sb} sb'
by (simp add: sb Prog_{sb})
from valid-reads-nth-update [OF i-bound this]
show ?thesis by (simp add: m_{sb}' ts_{sb}' \osb ')
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: instrs (dropWhile (Not \circ is-volatile-Write_{sb}) sb') \osb is_{sb} =
is @ prog-instrs (dropWhile (Not \circ is-volatile-Write_{sb}) sb')
by (simp add: suspends sb Prog_{sb})

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 Prog_{sb}]]
note share-commute =
  share-all-until-volatile-write-update-sb [of sb' sb, OF - i-bound ts_{sb}-i, simplified sb
  Prog_{sb}, simplified]

have (ts_{sb} [i := (p_{sb}, i_{sb}, sb', D_{sb}, \emptyset_{sb}, \mathcal{R}_{sb}, \mathcal{S}_{sb}), m_{sb}, \mathcal{S}_{sb}']) \sim (ts, m, \mathcal{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'
apply (clarsimp simp add: Let-def nth-list-update sb suspends Prog_{sb}
  R_{sb}', \mathcal{S}_{sb}')
apply (clarsimp simp add: nth-list-update sb suspends Prog_{sb}
  R_{sb}', \mathcal{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-sharing'
valid-program-history' valid'
apply (clarsimp simp add: m flush-commute)
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'
t_{sb}', \mathcal{S}_{sb}', \mathcal{O}_{sb}', \mathcal{R}_{sb}', \mathcal{S}_{sb}'
by (auto simp del: fun-upd-apply)
next
case (Ghost_{sb} A L R W)
from flush Ghost_{sb}
obtain
\mathcal{O}_{sb}': \mathcal{O}_{sb} = \mathcal{O}_{sb} \cup A - R and
\mathcal{S}_{sb}': \mathcal{S}_{sb} = \mathcal{S}_{sb} \oplus W R \ominus A L and
\mathcal{R}_{sb}': \mathcal{R}_{sb} = \text{augment-rels (dom } \mathcal{S}_{sb}) R \mathcal{R}_{sb} and
m_{sb}': m_{sb} = m_{sb}
by (auto simp add: sb)

from sharing-consis [OF i-bound ts_{sb}-i]
obtain
A-shared-owned: A \subseteq \text{dom } \mathcal{S}_{sb} \cup \mathcal{O}_{sb} and
L-subset: L \subseteq A and
A-R: A \cap R = \{\} and
R-owned: R \subseteq \mathcal{O}_{sb}
by (clarsimp simp add: sb Ghost_{sb})

have valid-own': valid-ownership \mathcal{S}_{sb}', t_{sb}'
proof (intro-locales)
show outstanding-non-volatile-refs-owned-or-read-only \mathcal{S}_{sb}', t_{sb}'
proof
fix j is \_j R_j D_j acq \_j sb_j p_j
assume j-bound: j < \text{length } t_{sb}'
assume t_{sb}'-j: t_{sb}' j = (p_j, i_{sb}, \_j sb_j D_j, \mathcal{O}_j, R_j)
show non-volatile-owned-or-read-only False \mathcal{S}_{sb}' \mathcal{O}_{sb} 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 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}' sb' O_{sb} ')

next

case False

from j-bound **have** j-bound': \( j < \) 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,xs_j,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 \( \cap \) 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-unowned \( (S_{sb} \oplus W R \ominus A L) \)

(\( \) acquired True \( (\) takeWhile \( (\) Not \( o \) is-volatile-Write_{sb} \) sb_j \) \( O_j \))

by (auto simp add: in-read-only-convs sb Ghost sb)

a \( \in \) read-only \( S_{sb} \) \( \longrightarrow a \in \) read-only \( (S_{sb} \oplus W R \ominus A L) \)

by (auto simp add: ts_{sb} ')

**from** outstanding-volatile-writes-unowned-by-others \( [\) OF i-bound j-bound' ts_{sb}']

 qed

qed

next

**show** outstanding-volatile-writes-unowned-by-others ts_{sb}'

**proof** (unfold-locales)

fix i_j p_i is_i O_i R_i D_i xs_i sb_i 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_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 \( i_1=1 \))

case True

**from** i_1-j True **have** neq-i-j: \( i \neq j \)

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

671
\texttt{ts}_{sb}^{-i} \texttt{ts-j} \uparrow \texttt{ts-i}_1 \text{ i-bound} \texttt{ts}_{sb}^{-i} \text{ True} \texttt{show} \ ?\text{thesis} \\
\textup{by} \ (\text{clarsimp simp add: ts}_{sb}^{-i} \texttt{sb} \text{ Ghost}_{sb}) \\
\text{next} \\
\text{case} \ False \\
\text{note} \ i_1-i = \text{this} \\
\text{from} \ i_1\text{-bound} \ \texttt{have} \ i_1\text{-bound}^{'}: i_1 < \text{length} \ \texttt{ts}_{sb} \\
\text{by} \ (\text{simp add: ts}_{sb}^{-i} \texttt{sb}) \\
\text{hence} \ i_1\text{-bound}^{''}: i_1 < \text{length} \ (\text{map} \ \text{owned} \ \texttt{ts}_{sb}) \\
\text{by} \ \text{auto} \\
\text{from} \ \texttt{ts-i}_1 \ \texttt{False} \ \text{have} \ \texttt{ts-i}_1^{'}: \texttt{ts}_{sb}!i_1 = (p_1, is_1, x_1, sb_1, D_1, O_1, R_1) \\
\text{by} \ (\text{simp add: ts}_{sb}^{-i} \texttt{sb}) \\
\text{show} \ ?\text{thesis} \\
\text{proof} \ (\text{cases} \ j=i) \\
\text{case} \ True \\
\text{from} \ \text{outstanding-volatile-writes-unowned-by-others} [\text{OF} \ i_1\text{-bound}^{'} \text{-bound} i_1-i \ \text{ts-i}_1^{'} \ \texttt{ts}_{sb}^{-i}] \\
\text{have} \ (O_{sb} \cup \text{all-acquired sb}) \cap \text{outstanding-refs} \ \text{is-volatile-Write}_{sb} \ sb_1 = \{\}. \\
\text{then} \ \text{show} \ ?\text{thesis} \\
\text{using} \ True \ i_1-i \ \texttt{ts-j} \ \texttt{ts}_{sb}^{-i} \ \text{i-bound} \\
\text{by} \ (\text{auto simp add: sb \ Ghost}_{sb} \texttt{ts}_{sb}^{-i} O_{sb}^{'} ) \\
\text{next} \\
\text{case} \ False \\
\text{from} \ j\text{-bound} \ \texttt{have} \ j\text{-bound}^{'}: j < \text{length} \ \texttt{ts}_{sb} \\
\text{by} \ (\text{simp add: ts}_{sb}^{-i} ) \\
\text{from} \ \texttt{ts-j} \ \texttt{False} \ \text{have} \ \texttt{ts-j}^{'}: \texttt{ts}_{sb}!j = (p_j, is_j, x_j, sb_j, D_j, O_j, R_j) \\
\text{by} \ (\text{simp add: ts}_{sb}^{-i} ) \\
\text{from} \ \text{outstanding-volatile-writes-unowned-by-others} [\text{OF} \ i_1\text{-bound}^{'} \text{-bound}^{'} i_1-j \ \text{ts-i}_1^{'} \ \texttt{ts-j}^{'}] \\
\text{show} \ (O_j \cup \text{all-acquired sb}_j) \cap \text{outstanding-refs} \ \text{is-volatile-Write}_{sb} \ sb_1 = \{\}. \\
\text{qed} \\
\text{qed} \\
\text{next} \\
\text{show} \ \text{read-only-reads-unowned} \ \texttt{ts}_{sb}^{-}\prime \\
\text{proof} \\
\text{fix} \ n \ m \\
\text{fix} \ p_n \ is_n \ O_n \ R_n \ D_n \ \emptyset_n \ sb_n \ p_m \ is_m \ O_m \ R_m \ D_m \ \emptyset_m \ sb_m \\
\text{assume} \ \text{n-bound}: \ n < \text{length} \ \texttt{ts}_{sb}^{-}\prime \\
\text{and} \ \text{m-bound}: \ m < \text{length} \ \texttt{ts}_{sb}^{-}\prime \\
\text{and} \ \text{neq-n-m}: \ n \neq m \\
\text{and} \ \text{nth}: \texttt{ts}_{sb}^{-}\prime n = (p_n, is_n, \emptyset_n, sb_n, D_n, O_n, R_n) \\
\text{and} \ \text{nth}: \texttt{ts}_{sb}^{-}\prime m = (p_m, is_m, \emptyset_m, sb_m, D_m, O_m, R_m) \\
\text{from} \ \text{n-bound} \ \texttt{have} \ \text{n-bound}^{'}: n < \text{length} \ \texttt{ts}_{sb}^{-}\prime \ \text{by} \ (\text{simp add: ts}_{sb}^{-}) \\
\text{from} \ \text{m-bound} \ \texttt{have} \ \text{m-bound}^{'}: m < \text{length} \ \texttt{ts}_{sb}^{-}\prime \ \text{by} \ (\text{simp add: ts}_{sb}^{-}) \\
\text{show} \ (O_{m} \cup \text{all-acquired sb}_{m}) \cap \ \\
\text{read-only-reads} \ (\text{acquired} \ True \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb} \ sb_n) \ O_n) \\
(\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb} \ sb_n)) = \{\} \\
\text{proof} \ (\text{cases} \ m=i) \\
\text{case} \ \text{True}
with neq-n-m have neq-n-i: \(n \neq i\)
by auto

with n-bound nth i-bound have nth': \(ts_{sb}!n = (p_n, is_n, \emptyset_n, sb_n, D_n, O_n, R_n)\)
by (auto simp add: \(ts_{sb}'\))
note read-only-reads-unowned [OF n-bound i-bound neq-n-i nth ts_{sb}]
then
show ?thesis
using True ts_{sb}-i neq-n-i nth mth n-bound
by (auto simp add: ts_{sb}′)

next
case False
note neq-m-i = this
with m-bound mth i-bound have mth': \(ts_{sb}!m = (p_m, is_m, \emptyset_m, sb_m, D_m, O_m, R_m)\)
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 \((\text{Not} \circ \text{is-volatile-Write}_{sb})\) \(sb_n)\)
(dropWhile \((\text{Not} \circ \text{is-volatile-Write}_{sb})\) \(sb_n\))]

have read-only-reads
(acquired True (takeWhile \((\text{Not} \circ \text{is-volatile-Write}_{sb})\) \(sb_n\)) \(O_{sb} \cup A - R\))
(dropWhile \((\text{Not} \circ \text{is-volatile-Write}_{sb})\) \(sb_n\)) \(\subseteq\) read-only-reads \((O_{sb} \cup A - R)\)

sb_n
by auto

with ts_{sb}-i nth mth neq-m-i n-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

next
show ownership-distinct 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: \(i_1 < \text{length } ts_{sb}'\)
assume j-bound: \(j < \text{length } ts_{sb}'\)
assume i_1\_j: \(i_1 \neq j\)
assume ts-i_1: ts_{sb}' \(\forall i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1)\)
\textbf{assume} \( \text{ts-j}: \text{ts}_{sb}!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)

\textbf{show} \((O_1 \cup \text{all-acquired sb}_1) \cap (O_j \cup \text{all-acquired sb}_j) = \{\} \)

\textbf{proof} (cases \( i_1 = i \))

\textbf{case} True

\textbf{with} \( i_1-j \) \textbf{have} \( i-j: i \neq j \)

\hspace{1em} \textbf{by simp}

\textbf{from} \( j \)-bound \textbf{have} \( j \)-bound': \( j < \text{length ts}_{sb} \)

\hspace{1em} \textbf{by (simp add: ts}_{sb}' \)

\textbf{hence} \( j \)-bound'': \( j < \text{length (map owned ts}_{sb} \)

\hspace{1em} \textbf{by simp}

\textbf{from} \( ts-j \) \( i-j \) \textbf{have} \( ts-j': ts_{sb}!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)

\hspace{1em} \textbf{by (simp add: ts}_{sb}' \)

\textbf{from} ownership-distinct \( [\text{OF } i \)-bound \( j \)-bound' \( i-j \) ts}_{sb}-i ts-j' \)

\textbf{show} \( ?\text{thesis} \)

\hspace{1em} \textbf{using} \( ts_{sb}-i \) True \( ts-i \) \( i \)-bound \( O)_{sb}' \)

\hspace{1em} \textbf{by (auto simp add: ts}_{sb}' sb Ghost_{sb} \)

\textbf{next}

\textbf{case} False

\textbf{note} \( i_1-i = \) this

\textbf{from} \( i_1 \)-bound \textbf{have} \( i_1 \)-bound': \( i_1 < \text{length ts}_{sb} \)

\hspace{1em} \textbf{by (simp add: ts}_{sb}' \)

\textbf{hence} \( i_1 \)-bound'': \( i_1 < \text{length (map owned ts}_{sb} \)

\hspace{1em} \textbf{by simp}

\textbf{from} \( ts-i_1 \) False \textbf{have} \( ts-i_1': ts_{sb}!i_1 = (p_1, is_1, xs_1, sb_1, D_1, O_1, R_1) \)

\hspace{1em} \textbf{by (simp add: ts}_{sb}' \)

\textbf{show} \( ?\text{thesis} \)

\textbf{proof} (cases \( j = i \))

\textbf{case} True

\textbf{from} ownership-distinct \( [\text{OF } i_1 \)-bound' \( i \)-bound' \( i_1 \) ts-i_1' ts}_{sb}-i \)

\textbf{show} \( ?\text{thesis} \)

\hspace{1em} \textbf{using} \( ts_{sb}-i \) True \( ts-j \) \( i \)-bound \( O)_{sb}' \)

\hspace{1em} \textbf{by (auto simp add: ts}_{sb}' sb Ghost_{sb} \)

\textbf{next}

\textbf{case} False

\textbf{from} \( j \)-bound \textbf{have} \( j \)-bound': \( j < \text{length ts}_{sb} \)

\hspace{1em} \textbf{by (simp add: ts}_{sb}' \)

\textbf{from} \( ts-j \) False \textbf{have} \( ts-j': ts_{sb}!j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j) \)

\hspace{1em} \textbf{by (simp add: ts}_{sb}' \)

\textbf{from} ownership-distinct \( [\text{OF } i_1 \)-bound' \( j \)-bound' \( i_1 \) ts-i_1' ts-j' \)

\textbf{show} \( ?\text{thesis} \)

\hspace{1em} \textbf{qed}

\hspace{1em} \textbf{qed}

\hspace{1em} \textbf{qed}

\hspace{1em} \textbf{have} valid-sharing': \( \text{valid-sharing (S}_{sb} \oplus W R \ominus A L) ts_{sb}' \)

\textbf{proof} (intro-locals)

\textbf{show} \( \text{outstanding-non-volatile-writes-unshared (S}_{sb} \oplus W R \ominus A L) ts_{sb}' \)
proof (unfold-locales)
  fix j p j is \( j \in \mathcal{O}_j \mathcal{R}_j \mathcal{D}_j \) acq \( j \in \mathcal{O}_j \mathcal{R}_j \mathcal{D}_j \) sb
  assume j-bound: \( j < \text{length ts}_{sb} \)
  assume jth: \( \text{ts}_{sb}'! j = (p_j, is_j, xs_j, sb, \mathcal{O}_j, \mathcal{R}_j) \)
  show non-volatile-writes-unshared \( (\mathcal{S}_{sb} \oplus \mathcal{W} R \ominus A) \) sb

proof (cases \( i=j \))
  case True
   with outstanding-non-volatile-writes-unshared [OF i-bound ts_{sb}-i]
   i-bound ts_{sb}-i show \{ thesis
   by (clarsimp simp add: ts_{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, xs_j, sb, \mathcal{O}_j, \mathcal{R}_j)
  by (auto simp add: ts_{sb}')
  from j-bound jth i-bound False
  have j: non-volatile-writes-unshared \( \mathcal{S}_{sb} \) sb
   apply −
   apply (rule outstanding-non-volatile-writes-unshared)
   apply (auto simp add: ts_{sb}')
   done
  from jth False have jth': ts_{sb}! j = (p_j, is_j, xs_j, sb, \mathcal{O}_j, \mathcal{R}_j)
  by (auto simp add: ts_{sb}')
  from outstanding-non-volatile-writes-unshared [OF j-bound' jth']
  have unshared: non-volatile-writes-unshared \( \mathcal{S}_{sb} \) sb

  have \( \forall a \in \text{dom (} \mathcal{S}_{sb} \oplus \mathcal{W} R \ominus A \) \) − \( \text{dom } \mathcal{S}_{sb} \), a \( \notin \) outstanding-refs is-non-volatile-Write_{sb} sb
   proof −
   { fix a
   assume a-in: a \( \in \) dom \( (\mathcal{S}_{sb} \oplus \mathcal{W} R \ominus A \) \) − dom \( \mathcal{S}_{sb} \)
   hence a-R: a \( \in R \)
   by clarsimp
   assume a-in-j: a \( \in \) outstanding-refs is-non-volatile-Write_{sb} sb
   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
    a-in-j
    have a \( \in \mathcal{O}_j \cup \) all-acquired sb
    by auto

moreover
  with ownership-distinct [OF i-bound j-bound' False ts_{sb}-i jth'] a-R R-owned
  show False
  by blast
qed
}
thus \( \text{thesis} \) by blast

qed

from non-volatile-writes-unshared-no-outstanding-non-volatile-Write_{sb}
[OF unshared this]
show \( \text{thesis} \).
qed
qed

next
show sharing-consis \((S_{sb} \oplus W \ominus A L) \ts_{sb}'\)

proof (unfold-locales)
fix \( j \) \( p_j \) is \( j \) \( R_j \) \( D_j \) acq \( j \) \( x_j \) sb_j
assume \( j \)-bound: \( j < \) length \( ts_{sb}' \)
assume \( j \)th: \( ts_{sb}' ! j = (p_j, is_j, x_j, sb_j, D_j, O_j, R_j) \)
show sharing-consistent \((S_{sb} \oplus W \ominus A L) \ O_j \ sb_j \)

proof (cases i=j)

case True
with \( j \)th \( ts_{sb}-i \) sharing-consis [OF \( i \)-bound \( ts_{sb}-i \)]
show \( \text{thesis} \)
by (clarsimp simp add: \( ts_{sb}' \) sb Ghost \( sb \) \( O_{sb}' \))

next
case False
from \( j \)-bound have \( j \)-bound': \( j < \) length \( ts_{sb} \)
by (auto simp add: \( ts_{sb}' \))
from \( j \)th False have \( j \)th': \( ts_{sb} ! j = (p_j, is_j, x_j, sb_j, D_j, O_j, R_j) \)
by (auto simp add: \( ts_{sb}' \))
from sharing-consis [OF \( j \)-bound' \( j \)th']
have consis: sharing-consistent \( S_{sb} \ O_j \ sb_j \).

have acq-cond: all-acquired \( sb_j \) \( \cap \) dom \( S_{sb} \) – dom \( S_{sb} \oplus W \ominus A L \) = \{\}

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 ownership-distinct [OF \( i \)-bound \( j \)-bound' \( \)False \( ts_{sb}-i \) \( j \)th']
have A \( \cap \) all-acquired \( sb_j \) = \{\}
by (auto simp add: sb Ghost_{sb})
with a-acq a-L L-subset

show False
by blast

qed

}  
thus \( \text{thesis} \)

by auto
have uns-cond: all-unshared \( s_b \cap \text{dom} \ (S_{sb} \oplus W \oplus A \ L) \) − \( \text{dom} \ S_{sb} = {} \)
proof −
{ 
fix a 
assume a-uns: \( a \in \text{all-unshared} \ s_b \) 
assume a \( \notin \ L \) 
assume a-R: \( a \in R \) 
have False 
  proof − 
from unshared-acquired-or-owned [OF consis] a-uns 
have a \( \in \text{all-acquired} \ s_b \cup O \) by auto 
with ownership-distinct [OF i-bound j-bound′ False ts_{sb}-i jth′] R-owned a-R 
show False 
  by blast 
qed 
} 
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 (\( S_{sb} \oplus W \oplus A \ L) \) ts_{sb}′ 
proof (unfold-locales) 
show − \( \bigcup ((\lambda (-,-,-,O,-,-).O) ^ i \set ts_{sb}′) \subseteq \text{dom} \ (S_{sb} \oplus W \oplus A \ L) \) 
proof − 

have s: \( \bigcup ((\lambda (-,-,-,O,-,-).O) ^ i \set ts_{sb}′) = \bigcup ((\lambda (-,-,-,O,-,-).O) ^ i \set ts_{sb}) \cup A - R \) 

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 \(j\) \(p_j\) \(is_j\) \(O_j\) \(R_j\) \(D_j\) \(xs_j\) \(sb_j\)
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 \(O_j \cap \text{read-only} \ (S_{sb} \oplus W R \ominus A L) = \{\}\)

proof (cases \(i=j\))
  case True
  from read-only-unowned [OF i-bound ts_{sb}-i]
  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}′ 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}′ \(! j = (p_j, is_j, xs_j, sb_j, D_j, O_j, R_j)\)
  by (auto simp add: ts_{sb}′)
from read-only-unowned [OF j-bound′ jth′]
  have \(O_j \cap \text{read-only} \ S_{sb} = \{\}.\)
  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 no-outstanding-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′ jth′]
have nw: no-write-to-read-only-memory $S_{sb}$ $sb_j$.

have $R \cap \text{outstanding-refs is-Write}_{sb} sb_j = \{\}$

proof
  - note dist = ownership-dist [OF i-bound $j$-bound False $ts_{sb}$-i $j$th]
  from non-volatile-owned-or-read-only-outstanding-non-volatile-writes
  [OF outstanding-non-volatile-refs-owned-or-read-only [OF $j$-bound' $j$th']]
  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] $j$th' $ts_{sb}$-i ]
  have outstanding-refs is-volatile-Write_{sb} sb_j \cap O_{sb} = \{\}
  by (auto simp add: misc-outstanding-refs-convs)
  ultimately have outstanding-refs is-Write_{sb} sb_j \cap O_{sb} = \{\}
  by (auto simp add: in-read-only-convs)

with R-owned
show ?thesis by blast
qed

have \forall a \in \text{outstanding-refs is-Write}_{sb} sb_j:
  a \in \text{read-only} (S_{sb} \oplus W R \ominus A L) \rightarrow a \in \text{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} \cup 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]
  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 suspends)

have (ts,m,Ś) ⇒ d^* (ts,m,Ś) 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 ORD θ j sb.
  j < length ts sb → i ≠ j →
  ts sb ! j = (p, is, θ, sb, D, O, R) →
  (all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∪
   all-unshared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∪
   all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)) ∩ (R ∪ L ∪ A) = {}
proof –
{    
  fix j p i sb j O j R j D j θ j sb x
assume j-bound: j < length ts sb
assume neq-i-j: i ≠ j
assume jth: ts sb ! j = (p j, is j, θ j, sb j, D j, O j, R j)
assume x-shared: x ∈ all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∪
  all-unshared (takeWhile (Not ◦ is-volatile-Write sb) sb) ∪
  all-acquired (takeWhile (Not ◦ is-volatile-Write sb) sb)
assume x-R-L-A: x ∈ R ∪ L ∪ A
have False
have 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) (dropWhile
    (Not ◦ is-volatile-Write sb) sb)]
  all-unshared-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb) (dropWhile
    (Not ◦ is-volatile-Write sb) sb)]
  all-acquired-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb) (dropWhile
    (Not ◦ is-volatile-Write sb) sb)]
  have x ∈ all-acquired sb ∪ O sb
  by auto
moreover
from x-R-L-A R-owned L-subset
have x ∈ all-acquired sb ∪ O sb

680
by (auto simp add: sb Ghost sb)
moreover
note ownership-distinct [OF i-bound j-bound neq-i-j ts sb-i jth]
ultimately show False by blast
qed
}
thus ?thesis by blast
qed

{ fix j p j is j O j R j φ j sb j x 
assume jth: ts sb-j = (p j, is j, φ j, sb j, O j, R j) 
assume j-bound: j < length ts sb 
assume neq: i ≠ j 
have release (takeWhile (Not ◦ is-volatile-Write sb) sb j) 
  (dom S sb ∪ R − L) R j 
  = release (takeWhile (Not ◦ is-volatile-Write sb) sb j) 
  (dom S sb) R j 
proof –
  { fix a 
    assume a-in: a ∈ all-shared (takeWhile (Not ◦ is-volatile-Write sb) sb j) 
    have (a ∈ (dom S sb ∪ R − L)) = (a ∈ dom S sb) 
    proof –
      from ownership-distinct [OF i-bound j-bound neq ts sb-i jth]
      have A-dist: A ∩ (O j ∪ all-acquired sb j) = {}
        by (auto simp add: sb Ghost sb)
    from all-shared-acquired-or-owned [OF sharing-consis [OF j-bound jth]] a-in 
    all-shared-append [of (takeWhile (Not ◦ is-volatile-Write sb) sb j)]
    (dropWhile (Not ◦ is-volatile-Write sb) sb j)] 
    have a-in: a ∈ O j ∪ all-acquired sb j 
      by auto 
    with ownership-distinct [OF i-bound j-bound neq ts sb-i jth]
    have a /∈ (O sb ∪ all-acquired sb) by auto
    with A-dist R-owned A-shared-owned L-subset a-in
    obtain a /∈ R and a /∈ L
      by fastforce 
    then show ?thesis by auto
    qed
  }
then 
show ?thesis 
  apply –
  apply (rule release-all-shared-exchange)
  apply auto
done

681
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.

note share-commute = share-all-until-volatile-write-Ghost sb-commute [OF (ownership-distinct ts sb) (sharing-consis S sb ts sb)]
i-bound ts sb - i [simplified sb Ghost sb] dist-R-L-A

have (ts sb ! i := (p sb, is sb, th sb, D sb, O sb ∪ A - R, augment-rels (dom S sb) R R sb), m sb, S sb') ∼ (ts, m, S)
apply (rule sim-config, intros)
apply (simp add: m flush-commute)
apply (clarsimp simp add: S 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 Ghost sb R sb ' S sb'
split: if-split-asm)
apply (rule conjI)
apply fastforce
apply clarsimp
apply (frule (2) release-commute)
apply clarsimp
apply auto
done

ultimately
show ?thesis
using valid-own' valid-hist' valid-reads' valid-sharing' temps-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' R sb')

qed

next
case (Program i p sb is sb th sb D sb O sb R sb p sb' mis)
then obtain
  ts sb' = ts sb [i := (p sb', is sb @ mis, th sb, sb@[Prog sb p sb p sb' mis], D sb, O sb, R sb)]

and
  i-bound: i < length ts sb and
  ts sb - i: ts sb ! i = (p sb, is sb, th sb, sb, D sb, O sb, R sb) and
  prog: th sb ⊢ p sb → p (p sb', mis) and
  S sb': S sb' = S sb and
  m sb': m sb = m sb
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

682
\(\forall i \leq \text{length ts}_{sb}\):

let \((p, \text{is}_{sb}, \theta, sb, D_{sb}, O_{sb}, R) = ts_{sb}! i\); suspends = dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb

in \(\exists is D.\) instrs suspends \@ is_{sb} = is \@ prog-instrs suspends \land

\(D_{sb} = (D \lor \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \emptyset) \land\)

\(ts ! i =\)

(hd-prog p suspends, is, 
\(\theta \mid (\text{dom} \theta - \text{read-tmps suspends}), (), D,\) acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \(O_{sb},\) release (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) (\text{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 \circ \text{is-volatile-Write}_{sb}) sb \@ dropWhile (Not \circ \text{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 \(D\) where
suspends: suspends = dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb and
is-sim: instrs suspends \@ is_{sb} = is \@ prog-instrs suspends and
\(D: D_{sb} = (D \lor \text{outstanding-refs is-volatile-Write}_{sb} sb \neq \emptyset) \land\)

\(ts-i: ts ! i =\)

(hd-prog p_{sb} suspends, is, 
\(\theta_{sb} \mid (\text{dom} \theta_{sb} - \text{read-tmps suspends}), (), D,\) acquired True ?take-sb \(O_{sb},\) release ?take-sb (\text{dom} \(S_{sb}\) \(R_{sb}\))

by (auto simp add: Let-def)

from prog-step-preserves-valid [OF i-bound ts_{sb}-i prog valid]
have valid': valid ts_{sb}'
by (simp add: ts_{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
–
from outstanding-non-volatile-refs-owned-or-read-only [OF i-bound ts_{sb}-i]
have non-volatile-owned-or-read-only False \(S_{sb} O_{sb} (sb@[\text{Prog}_{sb} p_{sb} p_{sb}' \text{mis}]\))
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}' S_{sb}')
qed
next
show outstanding-volatile-writes-unowned-by-others ts_{sb}'
proof –
have out: outstanding-refs is-volatile-Write_{sb} (sb@[\text{Prog}_{sb} p_{sb} p_{sb}' \text{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
  have read-only-reads-unowned ts\(_{sb}\)′
    proof
      −
      from read-only-reads-unowned-nth-update [OF i-bound ts\(_{sb}\)-i ro this]
      show ?thesis
        by (simp add: ts\(_{sb}\)′)
      qed
    next
      show ownership-distinct ts\(_{sb}\)′
        proof
          −
          from ownership-distinct-instructions-read-value-store-buffer-independent
          [OF i-bound ts\(_{sb}\)-i, where sb′ = (sb@[Prog \(_{sb}\) p\(_{sb}\) p\(_{sb}\)′ mis])]
          show ?thesis
            by (simp add: ts\(_{sb}\)′ all-acquired-append)
        qed
      qed
    from valid-last-prog [OF i-bound ts\(_{sb}\)-i]
    have last-prog: last-prog p\(_{sb}\) sb = p\(_{sb}\).
    from valid-hist′: valid-history program-step ts\(_{sb}\)′
      proof
        −
        from valid-history [OF i-bound ts\(_{sb}\)-i]
        have history-consistent \(_{sb}\) (hd-prog p\(_{sb}\) sb) sb.
        from history-consistent-append-Prog\(_{sb}\) [OF prog this last-prog]
        have hist-consis′: history-consistent \(_{sb}\) (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
        −
        have valid-reads m\(_{sb}\) ts\(_{sb}\)′
          proof
            −
            have valid-reads m\(_{sb}\) ts\(_{sb}\)′
              qed
        qed
      qed
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-locales)
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-locales)
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
by (simp add: ts)
next
from read-tmps-distinct [OF i-bound ts sb]
have distinct-read-tmps (sb@[Prog sb pb pb' mis])
by (simp add: distinct-read-tmps-append)
from read-tmps-distinct-nth-update [OF i-bound this]
show read-tmps-distinct ts
by (simp add: ts)
next
from load-tmps-read-tmps-distinct [OF i-bound ts sb-i]
distinct-load-tmps-prog-step [OF i-bound ts sb-i prog valid]
have load-tmps (is sb@mis) ∩ read-tmps (sb@[Prog sb pb pb' mis]) = {}
by (auto 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)
qed

have valid-dd': valid-data-dependency ts
proof
from data-dependency-consistent-instrs [OF i-bound ts sb-i]
have data-dependency-consistent-instrs (dom ₀ 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 ₀ sb) (is sb@mis)
load-tmps (is sb@mis) ∩ ∪ (fst ' write-sops (sb@[Prog sb pb pb' 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)
qed

have load-tmps-fresh': load-tmps-fresh ts
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 ₀ sb = {}
by (auto simp add: load-tmps-append)
from load-tmps-fresh-nth-update [OF i-bound this]
show ?thesis
by (simp add: ts)
qed

have enough-flushs': enough-flushs ts
proof

from clean-no-outstanding-volatile-Write\_sb [OF i-bound ts\_sb\_i]

have ¬ P\_sb \rightarrow 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: \( \forall \text{sop} \in \text{store-sops (is\_sb@mis)} \). valid-sop sop

by (auto simp add: store-sops-append)

from valid-write-sops [OF i-bound ts\_sb\_i]

have \( \forall \text{sop} \in \text{write-sops (sb@[Prog\_sb p\_sb p\_sb′ mis])} \). valid-sop sop

by (auto simp add: write-sops-append)

from valid-sops-nth-update [OF i-bound 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 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
by (simp)

from ts-i have ts-i: ts ! i = (p, is, θ, ().
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, is @ mis, θ, ()
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 ◦ 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)],
m sb, S sb’)
from
have (ts sb[i] := (p sb, is sb @ mis, θ sb, (), D, acquired True (takeWhile (Not ◦ 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: S S sb’ share-commute)

release-append-Prog sb release-append
split: if-split-asm)
done

ultimately show ?thesis
using valid-own’ valid-hist’ valid-reads’ valid-sharing’ tmtps-distinct’ m sb’
valid-dd’ valid-sops’ load-tmps-fresh’ enough-flushs’ valid-sharing’
valid-program-history’ valid’
\[ S_{sb}', ts_{sb}' \]

by (auto simp del: fun-upd-apply simp add: acquired-append-Prog
release-append-Prog sb release-append flush-all)

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 \circ is-volatile-Write_{sb}), simplified, OF volatile-r]

obtain a' v' sb'' sop A' L' R' W' where

sb-split: sb = takeWhile (Not \circ is-volatile-Write sb) sb @ Write_{sb} True a' sop' v' A' L' R' W' # sb''

and

drop: dropWhile (Not \circ is-volatile-Write_{sb}) sb = Write_{sb} 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_{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-append-Prog sb-commute [OF i-bound ts_{sb}-i]

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 \circ is-volatile-Write_{sb})]

have drop-app-comm:

(dropWhile (Not \circ is-volatile-Write_{sb}) (sb @ [Prog sb p_{sb} p_{sb} p_{sb} mis])) =

dropWhile (Not \circ is-volatile-Write_{sb}) sb @ [Prog sb p_{sb} p_{sb} mis]

by simp

note share-commute =

share-all-until-volatile-write-update-sb [OF share-append-Prog sb i-bound ts_{sb}-i]

from \mathcal{D}

have \mathcal{D}' : D_{sb} = (\mathcal{D} \lor outstanding-refs is-volatile-Write_{sb} (sb@[Prog sb p_{sb} p_{sb} mis]))

\neq \{ \}

by (auto simp: outstanding-refs-append)

have (ts_{sb} | i := (p_{sb}, i sb@mis, \emptyset_{sb}, sb@[Prog sb p_{sb} p_{sb} mis], D_{sb}, O_{sb}, R_{sb})],

689
\[ m_{sb}, S_{sb}' \sim (ts, m, S) \]

**apply** (rule sim-config, intros)

**apply** (simp add: m flush-commute)

**apply** (clarsimp simp add: \( S S_{sb}', \) share-commute)

**using** leq

**apply** simp

**using** i-bound i-bound’ ts-sim ts-i is-sim suspends’ [simplified suspends] \( D' \)

**apply** (clarsimp simp add: Let-def nth-list-update Prog sb drop-app-comm instrs-append prog-instrs-append read-tmps-append hd-prog-append-Prog sb acquired-append-Prog sb release-append-Prog sb release-append \( S_{sb}' \)

**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-sharing’

valid-program-history’ valid’

\( S_{sb}' \)’ts_{sb}'

**by** (auto simp del: fun-upd-apply)

**qed**

**qed**

**theorem** (in xvalid-program) concurrent-direct-steps-simulates-store-buffer-history-steps:

**assumes** step-sb: \((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow sbh^* (ts_{sb}', m_{sb}', S_{sb}')\)

**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** \( \bigwedge ts \ S \ m. (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S) \Rightarrow \) safe-reach-direct safe-delayed \((ts, m, S) \Rightarrow \)

\[ \Rightarrow \]

valid-ownership \( S_{sb}' ts_{sb}' \land valid-reads m_{sb}' ts_{sb}' \land valid-history program-step ts_{sb}' \land \]

valid-sharing \( S_{sb}' ts_{sb}' \land tmps-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 \]

\((\exists ts' m' S'). (ts, m, S) \Rightarrow d^* (ts', m', S') \land (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S'))\)

**using** step-sb valid-own valid-sb-reads valid-hist valid-sharing tmps-distinct valid-sops valid-dd load-tmps-fresh enough-flushs valid-program-history valid

690
proof (induct rule: converse-rtranclp-induct-sbh-steps)
  case refl thus ?case
  by auto
next
  case (step ts sb m sb S sb ts sb'' m sb'' S sb''
  note first = (ts sb, m sb, S sb) \Rightarrow sbh (ts sb'', m sb'', S sb'')
  note sim = (ts sb, m sb, S sb) \sim (ts, m, S)
  note safe-reach = (safe-reach-direct safe-delayed (ts, m, S))
  note valid-own = (valid-ownership S sb ts sb)
  note valid-reads = (valid-reads m sb ts sb)
  note valid-hist = (valid-history program-step ts sb)
  note valid-sharing = (valid-sharing S sb ts sb)
  note valid-sops = (valid-sops ts sb)
  note valid-dd = (valid-data-dependency ts sb)
  note load-tmps-fresh = (load-tmps-fresh ts sb)
  note enough-flushs = (enough-flushs ts sb)
  note valid-prog-hist = (valid-program-history ts sb)
  note valid = (valid ts sb)
  from concurrent-direct-steps-simulates-store-buffer-history-step [OF first valid-own valid-reads valid-hist valid-sharing tmps-distinct valid-sops valid-dd load-tmps-fresh enough-flushs valid-prog-hist valid sim safe-reach]
  obtain ts'' m'' S'' where
    valid-own'': (valid-ownership S sb'' ts sb'' and
    valid-reads'': (valid-reads m sb'' ts sb'' and
    valid-hist'': (valid-history program-step ts sb'' and
    valid-sharing'': (valid-sharing S sb'' ts sb'' and
    tmps-dist'': (tmps-distinct ts sb'' and
    valid-dd'': (valid-data-dependency ts sb'' and
    valid-sops'': (valid-sops ts sb'' and
    load-tmps-fresh'': (load-tmps-fresh ts sb'' and
    enough-flushs'': (enough-flushs ts sb'' and
    valid-prog-hist'': (valid-program-history ts sb'' and
    valid'': (valid ts sb'' and
    steps: (ts, m, S) \Rightarrow d^* (ts'', m'', S'') and
    sim: (ts sb'', m sb'', S sb'') \sim (ts'', 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''
    tmps-dist'' valid-sops'' valid-dd'' load-tmps-fresh'' enough-flushs'' valid-prog-hist'' valid'' ]
  obtain ts' m' S' where
    valid: (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-data-dependency ts sb'
    valid-sops ts sb' load-tmps-fresh ts sb' enough-flushs ts sb'
    valid-program-history ts sb' valid ts sb' and
    last: (ts'', m'', S'') \Rightarrow d^* (ts', m', S') and
\( \text{sim'}: (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S') \)

by blast

\textbf{note} steps also \textbf{note} last

\textbf{finally} show \textbf{?case}

\hspace{1cm} \textbf{using} valid sim'

by blast

qed

\textbf{sublocale} initial_{sb} \subseteq tmps-distinct ..

\textbf{locale} xvalid-program-progress = program-progress + xvalid-program

\textbf{theorem} (in xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-history-execution:

\hspace{1cm} \textbf{assumes} exec-sb: \( (ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{shb}^* (ts_{sb}', m_{sb}', S_{sb}') \)

\hspace{1cm} \textbf{assumes} init: initial_{sb} ts_{sb} S_{sb}

\hspace{1cm} \textbf{assumes} valid: valid ts_{sb}

\hspace{1cm} \textbf{assumes} sim: \( (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S) \)

\hspace{1cm} \textbf{assumes} safe: safe-reach-direct safe-free-flowing \( (ts, m, S) \)

\hspace{1cm} \textbf{shows} \( \exists ts' m' S' \). \((ts, m, S) \Rightarrow_d^* (ts', m', S') \wedge (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S') \)

\textbf{proof} –

\hspace{1cm} \textbf{from} init interpret ini: initial_{sb} ts_{sb} S_{sb} .

\hspace{1cm} \textbf{from} safe-free-flowing-implies-safe-delayed’ \{OF init sim safe\}

\hspace{1cm} \textbf{have} safe-delayed: safe-reach-direct safe-delayed \( (ts, m, S) \).

\hspace{1cm} \textbf{from} local.ini.valid-ownership-axioms have valid-ownership \( S_{sb} ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.valid-reads-axioms have valid-reads \( m_{sb} ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.valid-history-axioms have valid-history program-step \( ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.valid-sharing-axioms have valid-sharing \( S_{sb} ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.tmps-distinct-axioms have tmps-distinct \( ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.valid-sops-axioms have valid-sops \( ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.valid-data-dependency-axioms have valid-data-dependency \( ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.load-tmps-fresh-axioms have load-tmps-fresh \( ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.enough-flushs-axioms have enough-flushs \( ts_{sb} \).

\hspace{1cm} \textbf{from} local.ini.valid-program-history-axioms have valid-program-history \( ts_{sb} \).

\hspace{1cm} \textbf{from} concurrent-direct-steps-simulates-store-buffer-history-steps \{OF exec-sb \}

\hspace{1cm} \langle \text{valid-ownership} \ S_{sb} ts_{sb} \rangle

\hspace{1cm} \langle \text{valid-reads} \ m_{sb} ts_{sb} \rangle \langle \text{valid-history program-step} \ ts_{sb} \rangle

\hspace{1cm} \langle \text{valid-sharing} \ S_{sb} ts_{sb} \rangle \langle \text{tmps-distinct} \ ts_{sb} \rangle \langle \text{valid-sops} \ ts_{sb} \rangle

\hspace{1cm} \langle \text{valid-data-dependency} \ ts_{sb} \rangle \langle \text{load-tmps-fresh} \ ts_{sb} \rangle \langle \text{enough-flushs} \ ts_{sb} \rangle

\hspace{1cm} \langle \text{valid-program-history} \ ts_{sb} \rangle \langle \text{valid sim safe-delayed} \rangle

\hspace{1cm} \textbf{show} \textbf{?thesis by auto}

\hspace{1cm} \textbf{qed}

\textbf{lemma} \text{filter-is-Write}_{sb}-\text{Cons-Write}_{sb}: \text{filter is-Write}_{sb} \ volatile \ a \ sop \ v \ A \ L \ R \ W \#ys

692
\[ \exists rs rws. (\forall r \in \text{set } rs. \text{is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r) \land \\
\text{x}s=rs@\text{Write}_{ab} \text{ volatile } a \ \text{sop} v \ A \ L \ R \ W \# rws \land \text{y}s=\text{filter is-Write}_{ab} rws \]

**proof** (induct \(xs\))

**case** Nil **thus** ?case **by** simp

**next**

**case** (Cons \(x\) \(xs\))

**note** feq = (filter is-Write\(_{ab}\) (\(x\)#\(xs\)) = \text{Write}_{ab} \text{ volatile } a \ \text{sop} v \ A \ L \ R \ W \# \text{y}s)

**show** ?case

**proof** (cases \(x\))

**case** (\(\text{Write}_{ab} \text{ volatile}' a' \ \text{sop}' v' A' L' R' W'\))

**with** feq **obtain** \(\text{volatile}='=\text{volatile} a'=a \ v'=v \ \text{sop}'=\text{sop} A'=A \ L'=L \ R'=R \ W'=W\)

\(\text{y}s = \text{filter is-Write}_{ab} \text{ xs}\)

**by** auto

thus ?thesis

apply –

apply (rule-tac \(x=[]\) in exI)

apply (rule-tac \(x=xs\) in exI)

apply (simp add: \text{Write}_{ab})

done

**next**

**case** (\(\text{Read}_{ab} \text{ volatile}' a' t' v'\))

**from** feq **have** filter is-Write\(_{ab}\) \(xs = \text{Write}_{ab} \text{ volatile } a \ \text{sop} v \ A \ L \ R \ W\# ys\)

**by** (simp add: \text{Read}_{ab})

**from** Cons.hyps [OF this] **obtain** \(rs \ rws\) **where**

\(\forall r \in \text{set } rs. \text{is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r\) **and**

\(\text{x}s=rs \ @ \text{Write}_{ab} \text{ volatile } a \ \text{sop} v \ A \ L \ R \ W\# \text{y}s\) **and**

\(\text{y}s=\text{filter is-Write}_{ab} \text{ rws}\)

**by** clarsimp

then show ?thesis

apply –

apply (rule-tac \(x=\text{Read}_{ab} \text{ volatile}' a' t' v'\#rs\) in exI)

apply (rule-tac \(x=rws\) in exI)

apply (simp add: \text{Read}_{ab})

done

**next**

**case** (\(\text{Prog}_{ab} \ p_1 \ p_2 \ \text{mis}\))

**from** feq **have** filter is-Write\(_{ab}\) \(xs = \text{Write}_{ab} \text{ volatile } a \ \text{sop} v \ A \ L \ R \ W\# ys\)

**by** (simp add: \text{Prog}_{ab})

**from** Cons.hyps [OF this] **obtain** \(rs \ rws\) **where**

\(\forall r \in \text{set } rs. \text{is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r\) **and**

\(\text{x}s=rs \ @ \text{Write}_{ab} \text{ volatile } a \ \text{sop} v \ A \ L \ R \ W\# \text{rws}\) **and**

\(\text{y}s=\text{filter is-Write}_{ab} \text{ rws}\)

**by** clarsimp

then show ?thesis

apply –

apply (rule-tac \(x=\text{Prog}_{ab} \ p_1 \ p_2 \ \text{mis}\#rs\) in exI)

apply (rule-tac \(x=rws\) in exI)

apply (simp add: \text{Prog}_{ab})

done

**next**
case \( \text{Ghost}_{ab} A' L' R' W' \)

from \( \text{feq have filter is-Write}_{ab} \) \( x = \text{Write}_{ab} \) volatile a sop \( v \) A L R W \# \( y s \)
by \( \text{(simp add: Ghost}_{ab} ) \)

from \( \text{Cons.hyps [OF this] obtain rs rws where} \)
\( \forall r \in \text{set rs. is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r \) \( \text{and} \)
\( x s = rs @ \text{Write}_{ab} \) volatile a sop \( v \) A L R W\# \( r w s \) \( \text{and} \)
\( y s = \text{filter is-Write}_{ab} \) rws

by \( \text{clarsimp} \)
then show \(?\)thesis

apply –
apply \( \text{(rule-tac x= Ghost}_{ab} A' L' R' W' \# rs \) in \( \text{exI} ) \)
apply \( \text{(rule-tac x=rws in exI} ) \)
apply \( \text{(simp add: Ghost}_{ab} ) \)
done

qed

qed

lemma \( \text{filter-is-Write}_{ab}-\text{empty}: \) \( \text{filter is-Write}_{ab} \) \( x s = \[ \]

\( \Rightarrow \) \( \forall r \in \text{set xs. is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r \)

proof \( \text{(induct xs) } \)
case Nil thus \(?\)case by simp
next

\( \text{case (Cons x xs) } \)

\( \text{note feq = } \langle \text{filter is-Write}_{ab} \) \( (x \# xs) = \[ \rangle \)

\( \text{show } \) ?case
proof \( \text{(cases x) } \)

\( \text{case (Write}_{ab} \) volatile' \( a' v' \)

\( \text{with feq have False } \)

by \( \text{simp} \)

thus \(?\)thesis ..

next

\( \text{case (Read}_{ab} a' v' \)

from \( \text{feq have filter is-Write}_{ab} \) \( x s = \[ \)
by \( \text{(simp add: Read}_{ab} ) \)

from \( \text{Cons.hyps [OF this] obtain} \)
\( \forall r \in \text{set xs. is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r \)

by \( \text{clarsimp} \)
then show \(?\)thesis

by \( \text{(simp add: Read}_{ab} ) \)

next

\( \text{case (Prog}_{ab} p_2 p_2 \) mis \)

from \( \text{feq have filter is-Write}_{ab} \) \( x s = \[ \)
by \( \text{(simp add: Prog}_{ab} ) \)

from \( \text{Cons.hyps [OF this] obtain} \)
\( \forall r \in \text{set xs. is-Read}_{ab} r \lor \text{is-Prog}_{ab} r \lor \text{is-Ghost}_{ab} r \)

by \( \text{clarsimp} \)
then show \(?\)thesis

by \( \text{(simp add: Prog}_{ab} ) \)

next

\( \text{case (Ghost}_{ab} A' L' R' W' \)

694
from eq have filter is-Write sb xs = []
  by (simp add: Ghost sb)
from Cons.hyps [OF this] obtain
  \( \forall r \in \text{set}\, \text{xs}, \text{is-Read}_s\, r \lor \text{is-Prog}_s\, r \lor \text{is-Ghost}_s\, r \)
  by clarsimp
then show \( \? \text{thesis} \)
  by (simp add: Ghost sb)
qed
qed

lemma flush-reads-program:
  \[ \forall r \in \text{set}\, \text{sb}, \text{is-Read}_s\, r \lor \text{is-Prog}_s\, r \lor \text{is-Ghost}_s\, r = \Rightarrow \exists O' R' S'. \, (m, \text{sb}, O, R, S) \rightarrow_f (m, [], O', R', S') \]
proof (induct sb)
  case Nil thus \( \? \text{case} \) by auto
next
  case (Cons x sb)
  note \langle \forall r \in \text{set}\, (x \# \text{sb}), \text{is-Read}_s\, r \lor \text{is-Prog}_s\, r \lor \text{is-Ghost}_s\, r \rangle
  then obtain x: is-Read sb x \lor is-Prog sb x \lor is-Ghost sb x
    and sb: \forall r \in \text{set}\, \text{sb}, \text{is-Read}_s\, r \lor \text{is-Prog}_s\, r \lor \text{is-Ghost}_s\, r
    by (cases x) auto
  {
    assume is-Read sb x
    then obtain volatile a t v
      where x: x=Read sb volatile a t v
      by (cases x) auto
    have (m,Read sb volatile a t#sb, O, R, S) \rightarrow_f (m, sb, O, R, S)
      by (rule Read sb)
    also
    from Cons.hyps [OF sb] obtain O' S' acq' R'
      where (m, sb, O, R, S) \rightarrow_f^* (m, [], O', R', S')
      by blast
    finally
    have \( \? \text{case} \)
      by (auto simp add: x)
  }
moreover
{
  assume is-Prog sb x
  then obtain p1 p2 mis
    where x: x=Prog sb p1 p2 mis
    by (cases x) auto
  have (m,Prog sb p1 p2 mis#sb, O, R, S) \rightarrow_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) \rightarrow_f^* (m, [], O', R', S')
    by blast
  finally
  have \( \? \text{case} \)
}

by (auto simp add: x)
}
moreover
{
  assume is-Ghost
  then obtain A L R W where x: x=Ghost
  by (cases x) auto

  have (m,Ghost A L R W \#sb, O, R, S) \rightarrow f (m, sb, O \cup A - R, \text{augment-rels (dom } S) \ R R, S \oplus W R \ominus_A L)
    by (rule Ghost)
  also
  from Cons.hyps [OF sb] obtain O' S' R' acq'
    where (m, sb, O \cup A - R, \text{augment-rels (dom } S) \ R R, S \oplus W R \ominus_A L) \rightarrow 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: \exists m' O' S' R'. (m, r#sb, O, R, S) \rightarrow f (m', sb, O', R', S')
proof (cases r)
  case (Write sb volatile a sop v A L R W)
  from flush-step.Write [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)
next
  case (Read sb volatile a t v)
  from flush-step.Read [of m volatile a t v sb O R S]
  show ?thesis
    by (auto simp add: Read)
next
  case (Prog sb p1 p2 mis)
  from flush-step.Prog [of m p1 p2 mis sb O R S]
  show ?thesis
    by (auto simp add: Prog)
next
  case (Ghost sb 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) \rightarrow f^* (m', sb', O', R', S')
sows sb=[] \implies m'=m \land sb'=[] \land O'=O \land R'=R \land S'=S

696
using steps
apply (induct rule: converse-rtranclp-induct5)
apply (auto elim: flush-step.cases)
done

lemma flush-append:
assumes steps: \( (m, \text{sb}, O, R, S) \rightarrow t^* (m', \text{sb}', O', R', S') \)
shows \( \land_{\text{xs}} (m, \text{sb@xs}, O, R, S) \rightarrow t^* (m', \text{sb}'@\text{xs}, O', R', S') \)
using steps
proof (induct rule: converse-rtranclp-induct5)
case refl thus ?case by auto
next
case (step \( m \text{ sb} O R S \) \( m'' \text{ sb}'' O'' R'' S'' \) )
  note first = \( (m, \text{sb}, O, R, S) \rightarrow t (m'', \text{sb}'', O'', R'', S'') \)
  note rest = \( (m'', \text{sb}'' O'', R'', S'') \rightarrow t^* (m', \text{sb}', O', R', S') \)
  from step.hyps (3) have append-rest: \( (m'', \text{sb}''@\text{xs}, O'', R'', S'') \rightarrow t^* (m', \text{sb}'@\text{xs}, O', R', S') \).
from first show ?case
proof (cases)
case (Write\( a \text{ sb} \) volatile A R W L a sop v)
  then obtain \( \text{sb}: \text{sb}=\text{Write}_{a \text{ sb}} \) volatile a sop v A L R W\#sb'' and \( m'': m''=m(a=:v) \) and
  \( O'': O''=(\text{if volatile then } O \cup A - R \text{ else } O) \) and
  \( R'': R''=(\text{if volatile then } \text{Map.empty} \text{ else } R) \) and
  \( S'': S''=(\text{if volatile then } S \ominus_W R \ominus_A L \text{ else } S) \)
  by auto
have \( (m, \text{Write}_{a \text{ sb}} \) volatile a sop v A L R W\#sb''@\text{xs}, O, R, S) \rightarrow t \)
  \( (m(a=:v), \text{sb}'@\text{xs}, \text{if volatile then } O \cup A - R \text{ else } O, \text{if volatile then } \text{Map.empty} \text{ else } R, \)
  if volatile then \( S \ominus_W R \ominus_A L \text{ else } S) \)
  apply (rule flush-step.Write\( a \text{ sb} \))
  apply auto
done
hence \( (m, \text{sb}@\text{xs}, O, R, S) \rightarrow t (m'', \text{sb}'@\text{xs}, O'', R'', S'') \)
  by (simp add: \( \text{sb} m'' O'' R'' S'' \))
also note append-rest
finally show ?thesis .
next
case (Read\( a \text{ sb} \) volatile a t v)
then obtain \( \text{sb}: \text{sb}=\text{Read}_{a \text{ sb}} \) volatile a t v \#sb'' and \( m'': m''=m \)
  and \( O'': O''=O \) and \( S''=S \) and \( R'': R''=R \)
  by auto
have \( (m, \text{Read}_{a \text{ sb}} \) volatile a t v \#sb''@\text{xs}, O, R, S) \rightarrow t \)
  \( (m, \text{sb}'@\text{xs}, O, R, S) \)
  by (rule flush-step.Read\( a \text{ sb} \))
hence \( (m, \text{sb}@\text{xs}, O, R, S) \rightarrow t (m'', \text{sb}'@\text{xs}, O'', R'', S'') \)
  by (simp add: \( \text{sb} m'' O'' R'' S'' \))
also note append-rest
finally show ?thesis .
next
case (Prog\( a \text{ sb} p_1 p_2 \) mis)
then obtain \( \text{sb: sb=}\text{Prog}_{sb} p_1 p_2 \text{ mis#sb}'' \text{ and m}'' : \text{m}''=\text{m} \\
\text{and } \mathcal{O}'' : \mathcal{O}''=\mathcal{O} \text{ and } S'' : S''=S \text{ and } \mathcal{R}'' : \mathcal{R}''=\mathcal{R} \\
\text{by auto} \\
\text{have } (\text{m,Prog}_{sb} p_1 p_2 \text{ mis#sb}''@xs,\mathcal{O},\mathcal{R},S) \rightarrow_{\mathcal{I}} (\text{m, sb}''@xs,\mathcal{O},\mathcal{R},S) \\
\text{by (rule flush-step.Prog}_{sb} \\
\text{hence } (\text{m, sb}@xs,\mathcal{O},\mathcal{R},S) \rightarrow_{\mathcal{I}} (\text{m, sb}''@xs,\mathcal{O}''',\mathcal{R}'',S'') \\
\text{by (simp add: sb m}'' \text{ O}'' \mathcal{R}''' S''') \\
\text{also note append-rest} \\
\text{finally show } ?\text{thesis} . \\
\text{next} \\
\text{case (\text{Ghost A L R W})} \\
\text{then obtain } \text{sb: sb=}\text{Ghost}_{sb} A L R W\text{#sb}'' \text{ and m}'' : \text{m}''=\text{m} \\
\text{and } \mathcal{O}'' : \mathcal{O}''=\mathcal{O} \cup A \text{ - R and } S'' : S''=S \oplus W R \ominus A L \text{ and } \\
\mathcal{R}'' : \mathcal{R}''=\text{augment-rels } (\text{dom } S) R \mathcal{R} \\
\text{by auto} \\
\text{have } (\text{m, Ghost}_{sb} A L R W\text{#sb}''@xs,\mathcal{O},\mathcal{R},S) \rightarrow_{\mathcal{I}} (\text{m, sb}''@xs,\mathcal{O} \cup A \text{ - R, augment-rels} \\
(\text{dom } S) R \mathcal{R} S \oplus W R \ominus A L) \\
\text{by (rule flush-step.Ghost) } \\
\text{hence } (\text{m, sb}@xs,\mathcal{O},\mathcal{R},S) \rightarrow_{\mathcal{I}} (\text{m, sb}''@xs,\mathcal{O}''',\mathcal{R}'',S'') \\
\text{by (simp add: sb m}'' \text{ O}'' \mathcal{R}''' S''') \\
\text{also note append-rest} \\
\text{finally show } ?\text{thesis} . \\
\text{qed} \\
\text{qed} \\

\text{lemmas } \text{store-buffer-step-induct} = \\
\text{store-buffer-step.induct [split-format (complete),} \\
\text{consumes 1, case-names SBWrite}_{sb}] \\
\text{theorem } \text{flush-simulates-filter-writes:} \\
\text{assumes step: } (\text{m, sb, }\mathcal{O},\mathcal{R},S) \rightarrow_{w} (\text{m},'\text{sb}',\mathcal{O}',\mathcal{R}',S') \\
\text{shows } \bigwedge \text{sb}_{h}, \mathcal{O}_{h} R_{h} S_{h} : \text{sb}=\text{filter is-Write}_{sb} \text{ sb}_{h} \\
\Rightarrow \\
\exists \text{sb}''_{h} \mathcal{O}'_{h} R'_{h} S'_{h} : (\text{m, sb}_{h},\mathcal{O}_{h},R_{h},S_{h}) \rightarrow_{w} (\text{m},'\text{sb}''_{h},\mathcal{O}'_{h},R'_{h},S'_{h}) \land \\
\text{sb}''_{h}=\text{filter is-Write}_{sb} \text{ sb}_{h}'' \land (\text{sb}''=[]) \rightarrow (\text{sb}_{h}''=[]) \\
\text{using step} \\
\text{proof (induct rule: store-buffer-step-induct) } \\
\text{case } (\text{SBWrite}_{sb} \text{ m volatile a D f v A L R W sb } \mathcal{O} \mathcal{R} S) \\
\text{note } \text{filter-Write}_{sb} = (\text{Write}_{sb} \text{ volatile a (D,f) v A L R W# sb = filter is-Write}_{sb} \text{ sb}_{h}) \\
\text{from } \text{filter-is-Write}_{sb}-\text{Cons-Write}_{sb} [\text{OF filter-Write}_{sb} [\text{symmetric}]] \\
\text{obtain \text{rs where} } \\
\text{rs-reads: } \forall r \text{ rset rs. is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r \lor \text{and} \\
\text{sb}_{h}: \text{sb}_{h} = r @ \text{Write}_{sb} \text{ volatile a (D,f) v A L R W# rs and} \\
\text{sb: sb = filter is-Write}_{sb} \text{ rs} \\
\text{by blast} \\
\text{from } \text{flush-reads-program} [\text{OF rs-reads} \text{ obtain } \mathcal{O}_{h}' R_{h}' S_{h}' \text{ acq}_{h}' \\
\text{where } (\text{m, } \mathcal{O}_{h} R_{h} S_{h}) \rightarrow_{w} (\text{m},'[],\mathcal{O}_{h}',R_{h}',S_{h}') \text{ by blast} \\
\text{from } \text{flush-append} [\text{OF this}] \\

698
have \((m, \text{rs}@\text{Write}_{sb} \text{ volatile a (D,f)} v A L R W\# \text{ rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \rightarrow t^*\)

\((m, \text{Write}_{sb} \text{ volatile a (D,f)} v A L R W\# \text{ rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \)

by simp

also

from flush-step.\text{Write}_{sb} [\text{OF refl refl refl, of m volatile a (D,f) v A L R W rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h]\)

obtain \(\mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h\)

where \((m, \text{Write}_{sb} \text{ volatile a (D,f)} v A L R W\# \text{ rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \rightarrow t (m(a:=v), \text{rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h)\)

by auto

finally have steps: \((m, \text{sb,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \rightarrow t^* (m(a:=v), \text{rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h)\)

by (simp add: \text{sb})

show ?thesis

proof (cases \text{sb})

case Cons

with steps \text{sb} show ?thesis

by fastforce

next

case Nil

from filter-is-Write_{sb}-empty [\text{OF \text{sb [simplified Nil, symmetric]}]}]

have \(\forall r \in \text{rws. is-Read}_{sb} r \lor \text{is-Prog}_{sb} r \lor \text{is-Ghost}_{sb} r\)

from flush-reads-program [\text{OF this]} obtain \(\mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h\)

where \((m(a:=v), \text{rws,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \rightarrow t^* (m(a:=v), [], \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h)\) by blast

with steps

have \((m, \text{sb,} \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h) \rightarrow t^* (m(a:=v), [], \mathcal{O}_h, \mathcal{R}_h, \mathcal{S}_h)\) by force

with \text{sb Nil} show ?thesis by fastforce

qed

qed

lemma bufferd-val-filter-is-Write_{sb}-eq-ext:

buffered-val (filter is-Write_{sb} \text{ sb}) a = buffered-val \text{ sb a}

by (induct \text{ sb}) (auto split: memref.splits)

lemma bufferd-val-filter-is-Write_{sb}-eq:

buffered-val (filter is-Write_{sb} \text{ sb}) = buffered-val \text{ 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} \text{ xs}) =

outstanding-refs is-volatile-Write_{sb} \text{ xs}

by (induct \text{ xs}) (auto simp add: is-volatile-Write_{sb}-def split: memref.splits)

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: \(ts \sim_{h} ts_h\)

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 –

699
interpret sbh-computation:
  computation sbh-memop-step flush-step program-step
  \( \lambda p \, p' \) is sb. sb @ [Prog sb p p']

from step-sb
show ?thesis

proof (cases rule: concurrent-step-cases)
  case (Memop i - p is \( \theta \) sb \( D \, O \, R \) - - is \( \theta' \) sb' - \( D' \, O' \, R' \))
  then obtain
    ts': ts' = ts[i := (p, is', \( \theta' \), sb', \( D' \), \( O' \), \( R' \))]
    i-bound: i < length ts
    ts-i: ts ! i = (p, is, \( \theta \), sb, \( D \), \( O \), \( R \))
    step-sb: (is, \( \theta \), sb, m, \( D \), \( O \), \( R \), \( S \)) \( \rightarrow \) sb
      (is', \( \theta' \), sb', m', \( D' \), \( O' \), \( R' \), \( S' \))
  by auto

from sim obtain
  lts-eq: length ts = length ts_h
  sim-loc: \( \forall \) i < length ts. (\( \exists \) O' D' R'.
    let (p, is, \( \theta \), sb, D, O, R) = ts_i in
    ts_i = (p, is, \( \theta \), filter is-Write sb sb, D', O', R') \wedge
    (filter is-Write sb sb = [] \( \rightarrow \) sb = []))
  by cases (auto)

from lts-eq i-bound have i-bound': i < length ts_h
  by simp

from step-sb
show ?thesis

proof (cases)
  case (SBReadBuffered a v volatile t)
  then obtain
    is: is = Read volatile a t#is'
    O': O' = O
    S': S' = S
    R': R' = R
    D': D' = D
    m': m' = m
    \( \theta' \): \( \theta' = \theta(t \rightarrow v) \)
    sb': sb' = sb
    buf-val: buffered-val sb a = Some v
  by auto

from sim-loc [rule-format, OF i-bound] ts-i is
  sb_h O_h R_h D_h where
  ts_h-i: ts_h!i = (p, Read volatile a t#is', \( \theta \), sb_h, D_h, O_h, R_h)
  sb: sb = filter is-Write sb sb_h
  sb-empty: filter is-Write sb sb_h = [] \( \rightarrow \) sb_h = []
  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 sb)

let ?ts'h-i' = (p, is', \theta(t \mapsto v), sbh @ [Read sb volatile a t v], D_h, O_h, R_h)
let ?ts'h = ts'h[i := ?ts'h-i']
from sbh-memop-step.SBHReadBuffered [OF buf-val']
have (Read volatile a t # is', \theta, sbh, m, D_h, O_h, R_h, S_h) \rightarrow sbh
(is', \theta(t \mapsto v), sbh @ [Read sb volatile a t v], 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 sbh (?ts'h', 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', \theta(t \mapsto v), 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 ?thesis
  by (auto simp del: fun-upd-apply simp add: S' m' ts' O' \theta' D' sb' R')

next
  case True

  with sb-empty have empty: sbh=[] by simp
  from i-bound' have ?ts'h'n = ?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', m, S_h)

  \Rightarrow sbh (ts'h[i := (p, is', \theta(t \mapsto v), []], D_h, O_h, R_h]), m, S_h)
  by (simp add: empty list-update-overwrite)

  with step have (ts'h, m, S_h) \Rightarrow sbh
  (ts'h[i := (p, is', \theta(t \mapsto v), []], D_h, O_h, R_h]), m, S_h)
  by force

  moreover
  have ts [i := (p, is', \theta(t \mapsto v), sb, D, O, R)] \sim_h ts'h[i := (p, is', \theta(t \mapsto v), []], 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: \(S'\) \(m'\) \(ts'\) \(O'\) \(\theta'\) \(D'\) \(sb'\) \(R'\))

qed

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

\(\theta'\): \(\theta' = \theta(t \mapsto m a)\) 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 sbh \(O_h\) \(R_h\) \(D_h\) where

\(ts_h\)-i: \(ts_h\)!i = (p, Read volatile a t#is', \(\theta\), sbh, \(D_h\), \(O_h\), \(R_h\)) and

sb: sb = filter is-Write\(sb\) sbh and

sb-empty: filter is-Write\(sb\) sbh = [[] \(\mapsto\) sbh = []]

by (auto simp add: Let-def)

from buf

have buf': buffered-val sbh a = None

by (simp add: bufferd-val-filter-is-Write\(sb\)-eq sb)

let \(ts_h\)-i' = (p, is', \(\theta\) \(t \mapsto m a\), sbh \(\@\) Read\(sb\) volatile a t (m a), \(D_h\), \(O_h\), \(R_h\))

let \(?ts_h'\) = \(?ts_h\)-i'

from sbh-memop-step.SBHReadUnbuffered [OF buf']

have (Read volatile a \(\#\) is', \(\theta\), sbh, \(m\), \(D_h\), \(O_h\), \(R_h\), \(S_h\)) \(\Rightarrow\) sbh

(is', \(\theta\) \(t \mapsto (m a)\), sbh \(\@\) 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: (\(ts_h\), \(m\), \(S_h\)) \(\Rightarrow\) sbh

(\(?ts_h'\), \(m\), \(S_h\)).

moreover

from sb have sb: sb = filter is-Write\(sb\) (sbh \(\@\) Read\(sb\) volatile a t (m a))

by simp

show \(\text{thesis}\)

proof (cases filter is-Write\(sb\) sbh = [])

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.intro)

using lts-eq

apply simp

using sim-loc i-bound i-bound' sb sb-empty False

702
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' R' D' \vartheta' 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', m, S_h)
    \Rightarrow sbh (ts_h[i := (p, is', \vartheta(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', \vartheta(t \mapsto m a)), [], D_h, O_h, R_h]), m, S_h)
    by force
moreover
  have ts [i := (p,is',\vartheta(t \mapsto m a),sb,D,O,R)] \sim_h ts_h[i := (p, is', \vartheta(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' sb empty
    apply (auto simp add: Let-def is)
    done
ultimately show ?thesis
by (auto simp del: fun-upd-apply simp add: S m' ts' O' \vartheta' D' sb' R')
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': O'=O and
    R': R'=R and
    S': S'=S and
    D': D'=D and
    m': m'=m and
    \vartheta': \vartheta'=\vartheta and
    sb': sb' = sb@[Write sb False a (D, f) (f \vartheta) A L R W]
  by auto

from sim-loc [rule-format, OF i-bound] ts-i
  obtain sb_h O_h R_h D_h where
  ts_h-i: ts_h!i = (p,Write False a (D,f) A L R W#is',\vartheta,sb_h,D_h,O_h,R_h) and
  sb: sb = filter is-Write sb sb_h
  by (auto simp add: Let-def is)
from sbh-memop-step.SBHWriteNonVolatile
have (Write False a (D, f) A L R W# is', ð, sbh, m, D_h, O_h, R_h, S_h) \rightarrow_{sbh}
(is', ð, sbh @ [Write_{sbh} False a (D, f) (f ð) A L R W], m, D_h, O_h, R_h, S_h).

from sbh-computation.Memop [OF i-bound' ts_h-i this]
have (ts_h, m, S_h) \Rightarrow_{sbh}
(ts_h[i := (p, is', ð, sbh @ [Write_{sbh} False a (D, f) (f ð) A L R W], m, D_h, O_h, R_h, S_h])],
m, S_h).

moreover have ts [i := (p, is', ð, sbh @ [Write_{sbh} False a (D, f) (f ð) A L R W], m, D_h, O_h, R_h, 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 ?thesis
by (auto simp add: S' m' ð' O' R' D' ts' sb')

next
case (SBWriteVolatile a D f A L R W)
then obtain is: is = Write True a (D, f) A L R W#is' and
O': O'=O and
R': R'=R and
S': S'=S and
D': D'=D and
m': m'=m and
ð': ð'=ð and
sb': sb'= sb@[Write_{sb} True a (D, f) (f ð) A L R W]
by auto

from sim-loc [rule-format, OF i-bound] ts-i is
obtain sbh O_h R_h D_h where
ts_h-i: ts_h[i := (p,Write True a (D, f) A L R W#is', ð, sbh, D_h, O_h, R_h)] and
sb: sb = filter is-Write_{sb} sbh
by (auto simp add: Let-def)

from sbh-computation.Memop [OF i-bound' ts_h-i SBHWriteVolatile]

have (ts_h, m, S_h) \Rightarrow_{sbh}
(ts_h[i := (p, is', ð, sbh @ [Write_{sbh} True a (D, f) (f ð) A L R W], True, O_h, R_h)],
m, S_h).

moreover have ts [i := (p, is', ð, sbh @ [Write_{sbh} True a (D, f) (f ð) A L R W], D, O, R)] \sim_{h}
ts_h[i := (p, is', ð, sbh @ [Write_{sbh} True a (D, f) (f ð) A L R W], True, O_h, R_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 ?thesis
by (auto simp add: ts O O' R R' S S' D D' M m' θ θ' sb sb')

next

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
θ': θ = θ and
sb: sb = [] and
sb': sb' = []
by auto

from sim-loc [rule-format, OF i-bound] ts-i sb is
obtain sbh O R D h where
ts_h': ts_h'[i := (p, Fence # is', θ, [], D_h, O_h, R_h)] and
sb: [] = filter is-Write sb
by (auto simp add: Let-def)

from filter-is-Write-empty [OF sb [symmetric]]
have ∀ r ∈ set sb_h. is-Read sb r ∨ is-Prog sb r ∨ is-Ghost sb r.

from flush-reads-program [OF this] obtain O' S' h' R h'
where flsh: (m, sb_h, O_h, R_h, S_h) →t' (m, [], O_h', R_h', S_h') by blast

let ?ts_h' = ts_h'[i := (p, Fence # is', θ, [], D_h, O_h', R_h')]
from sbh-computation.store-buffer-steps [OF fish i-bound' ts_h-i]
have (ts_h, m, S_h) ⇒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, Fence#is', θ, [], D_h, O_h', R_h')]
by simp
from sbh-computation.Memop [OF i-bound'' ts_h'-i SBFence] i-bound'
have (?ts_h', m, S_h') ⇒sbh (ts_h[i := (p, is', θ, [], False, O_h', Map.empty)], m, S_h')
by (simp)
finally
have (ts_h, m, S_h) ⇒sbh* (ts_h[i := (p, is', θ, [], False, O_h', Map.empty)], m, S_h').
moreover

have ts [i := (p, is', θ', [D, O, R])] \sim_h ts_h [i := (p, is', θ, []) \cdot False, O_h', 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

ultimately show \textit{?thesis}
by (auto simp add: ts' O' \theta' m' sb' D' S' R')

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': O' = O and
R': R' = R and
S': S' = S and
D': D' = D and
m': m' = m and
\theta': \theta' = \theta(t \mapsto m a) and
sb: sb = [] and
sb': sb' = [] and
cond: \neg cond (\theta(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
\texttt{ts_h': ts_h[i := (p, RMW a t (D, f) cond ret A L R W # is', \theta, sb_h, D_h, O_h, R_h)]}
and sb: [] = filter is-Write 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-\textit{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 \ast (m, [], O_h', R_h', S_h') by blast

let \texttt{ts_h' = ts_h[i := (p, RMW a t (D, f) cond ret A L R W # is', \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} \ast (?ts_h', m, S_h')
also

from i-bound' have i-bound'': i < length \texttt{ts_h'}

706
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′, θ, [], D_h, O_h′, R_h′)
by simp

note step= SBHRMWReadOnly [where cond=cond and θ=θ and m=m, OF cond ]
from sbh-computation.Memop [OF i-bound′ ts_h′-i step ] i-bound′
have (?ts_h′, m, S_h′) ⇒ sbh (ts_h[i := (p, is′, θ(t⇨m a), []], False, O_h′, Map.empty)], m, S_h′)
by (simp)
finally
have (ts_h, m, S_h) ⇒ sbh∗ (ts_h[i := (p, is′, θ(t⇨m a), []], False, O_h′, Map.empty)], m, S_h′).
moreover

have ts [i := (p,is′, θ(t⇨m a), []], D, O, R] ⇒h ts_h[i := (p, is′, θ(t⇨m a), []], False, O_h′, Map.empty])
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 ?thesis
by (auto simp add: ts′ O′ θ′ m′ sb′ D′ S′ R′)
next

case (SBWMRWWrite 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′: O′=O and
R′: R′=R and
S′: S′=S and
D′: D′=D and
m′: m′=m(a := f (θ(t⇨(m a)))) and
θ′: θ′=θ(t⇨ ret (m a) (f (θ(t⇨(m a)))))) and
sb: sb=[] and
sb′: sb′=[] and
cond: cond (θ(t⇨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 acc_h where

obtain sb_h O_h R_h D_h acc_h where

obtain sb_h O_h R_h D_h acc_h where

707
from filter-is-Write sb-empty [OF sb [symmetric]]

have \( \forall r \in \text{set } sb \) \( \text{is-Read } sb \ r \lor \text{is-Prog } sb \ r \lor \text{is-Ghost } sb \ r \).

from flush-reads-program [OF this] obtain \( O_{h}', S_{h}', R_{h}' \)

where \( \text{flsh}: (m, sb_{h}, O_{h}, R_{h}, S_{h}) \rightarrow t^* (m, [], O_{h}', R_{h}', S_{h}') \) by blast

let \(?t_{sh}' = t_{sh}[i := (p, \text{RMW } a t (D, f) \text{ cond ret } A L R W#is', \theta, [], D_{h}, O_{h}', R_{h}')] \)

from sbh-computation.store-buffer-steps [OF flsh i-bound' ts_{sh}'-i]

have \( (t_{sh}, m, S_{h}) \Rightarrow_{sbh^*} (?t_{sh}', m, S_{h}') \).

also

from i-bound' have i-bound'": \( i < \text{length } ?t_{sh}' \)

by auto

from i-bound' have \( t_{sh}'[i := (p, \text{RMW } a t (D, f) \text{ cond ret } A L R W#is', \theta, [], D_{h}, O_{h}', R_{h}')] \)

by simp

note step = SBHMRMWrite [where cond = cond and \( \theta = \theta \) and \( m = m \), OF cond]

from sbh-computation.Memop [OF i-bound' ts_{sh}'-i step ] i-bound'

have \( (?t_{sh}', m, S_{h}') \Rightarrow_{sbh} (t_{sh}[i := (p, is', \\
(\theta(t \mapsto \text{ret (m a)}) (f (\theta(t \mapsto (m a)))))), [], \text{False}, O_{h}' \cup A - R, \text{Map.empty}])\),

\( m(a := f (\theta(t \mapsto (m a))))), S_{h}' \oplus W R \ominus A L) \)

by (simp)

finally

have \( (t_{sh}, m, S_{h}) \Rightarrow_{sbh^*} (t_{sh}[i := (p, is', \\
(\theta(t \mapsto \text{ret (m a)}) (f (\theta(t \mapsto (m a)))))), [], \text{False}, O_{h}' \cup A - R, \text{Map.empty}])\),

\( m(a := f (\theta(t \mapsto (m a))))), S_{h}' \oplus W R \ominus A L) \).

moreover

have \( ts[i := (p, is', \theta(t \mapsto \text{ret (m a)}) (f (\theta(t \mapsto (m a)))))), [], D_{h}, O_{h}', R] \sim_{h} \\
\( t_{sh}[i := (p, is', \theta(t \mapsto \text{ret (m a)}) (f (\theta(t \mapsto (m a)))))), [], \text{False}, O_{h}' \cup A - R, \text{Map.empty}]) \)

apply (rule sim-history-config.intro)

using lts-eq

apply simp

using sim-loc i-bound' sb

apply (auto simp add: Let-def nth-list-update)

done

ultimately show \(?\text{thesis}\)

by (auto simp add: ts' \( O' \theta' m' \) sb' \( D' S' R' \))

next
case (SBGhost A L R W)

then obtain

is: \( \text{is = Ghost } A L R W#is' \) and 
\( O': O' = O \) and 
\( R': R' = R \) and 

708
\( S' \vdash S' = S \) and 
\( D' \vdash D' = D \) and 
\( m' = m \) and 
\( \theta' = \theta \) and 
\( sb' = sb \) 
by auto 

\[ \begin{align*}
\text{from sim-loc [rule-format, OF i-bound] ts-i is} & \\
\text{obtain sb}_h \ \mathcal{O}_h \ \mathcal{R}_h \ \mathcal{D}_h \ \text{where} & \\
ts_h[i] = (p, \text{Ghost A L R W# is', \theta, sb}_h, \mathcal{D}_h, \mathcal{O}_h, \mathcal{R}_h) \ \text{and} & \\
\text{sb: sb = filter is-Write}_{sb} \ \text{sb and} & \\
\text{sb-empty: filter is-Write}_{sb} \ \text{sb = [ ] \rightarrow sb}_h = [ ] & \\
\text{by (auto simp add: Let-def)} & \\
\text{let \( ?ts'_h \vdash i' = (p, \text{is}', \theta, \text{sb}, \mathcal{D}_h, \mathcal{O}_h, \mathcal{R}_h) \) \&} & \\
\text{let \( ?ts'_h = ts_h[i \vdash ?ts'_h-i'] \) \&} & \\
\text{note step = SBHGhost \&} & \\
\text{from sbh-computation.Memop [OF i-bound' ts_h-i step] i-bound' \&} & \\
\text{have step: (ts}_h, \ \mathcal{S}_h) \ \vdash \ \text{sbh} (\ ?ts'_h, \mathcal{M}, \ \mathcal{S}_h) & \\
\text{by (simp)} & \\
\text{from sb have sb: sb = filter is-Write}_{sb} \ (\text{sb}_h \ \text{[Ghost}_{sb} \ \text{A L R W})} & \\
\text{by simp} & \\
\text{show \( ?\)thesis} & \\
\text{proof (cases filter is-Write}_{sb} \ \text{sb}_h = [ ])} & \\
\text{case False} & \\
\text{have ts \[i \vdash (p, \text{is}', \theta, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R})] \ \sim_h \ ?ts'_h' & \\
\text{apply (rule sim-history-config.intros)} & \\
\text{using lts-eq} & \\
\text{apply simp} & \\
\text{using sim-loc i-bound i-bound' sb sb-empty False} & \\
\text{apply (auto simp add: Let-def nth-list-update)} & \\
\text{done} & \\
\text{with step show \( ?\)thesis} & \\
\text{by (auto simp del: fun-upd-apply simp add: \( S' \ \mathcal{M} \ \mathcal{S}' \ \mathcal{O}' \ \mathcal{D}' \ \theta' \ \mathcal{R}' \))} & \\
\text{next} & \\
\text{case True} & \\
\text{with sb-empty have empty: sb}_h = [ ] \ \text{by simp} & \\
\text{from i-bound' have ?ts_h \[i \vdash ?ts'_h-i' \] & \\
\text{by auto} & \\
\text{from sbh-computation.StoreBuffer [OF - this, simplified empty, simplified, OF - flush-step.Ghost, of m \ \mathcal{S}_h] i-bound'} & \\
\text{have (?ts'_h', \mathcal{M}, \mathcal{S}_h) \ \vdash \ \text{sbh} (ts}_h[i \vdash (p, \text{is}', \theta, [ ], \mathcal{D}_h, \mathcal{O}_h, \text{A} \cup \mathcal{R} , \text{augment-rels (dom } \mathcal{S}_h) \text{ R } \mathcal{R}_h)]}, & \\
\text{m, } \mathcal{S}_h \mathcal{W} \text{ R } \mathcal{A} \text{ L}) & \\
\text{by (simp add: empty)} & \\
\text{with step have (ts}_h, \mathcal{M}, \mathcal{S}_h) \ \vdash \ \text{sbh}^* & \\
\end{align*} \]

709
\[ (ts_h[i := (p, is', \varnothing, D_h, O_h \cup A \to R, \text{augment-rels}(\text{dom } S_h) R R_h)], m, S_h) \]

Moreover

have \( ts \left[ i := (p, is', \varnothing, sb, D, O, R) \right] \sim_h \)

\[ ts_h[i := (p, is', \varnothing, D_h, O_h \cup A \to R, \text{augment-rels}(\text{dom } S_h) R R_h)] \]

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' \( \theta' D' sb' R' \))

qed

next

case (Program \( i - p \) is \( \theta \) sb \( D \ O \ R \) p' is')

then obtain

\( ts' : ts' = ts[i := (p', is@is', \varnothing, sb, D, O, R)] \) and

i-bound: \( i < \text{length } ts \) and

\( ts-i : ts[i = (p, is, \varnothing, sb, D, O, R)] \) and

prog-step: \( \theta\vdash p \to p'(p', is') \) and

\( S' : S' = S \) and

\( m' : m' = m \)

by auto

from sim obtain

lts-eq: length ts = length ts_h and

sim-loc: \( \forall i < \text{length } ts \quad (\exists O' D' R'). \)

let \( (p, is, \varnothing, sb, D, O, R) = ts_h[i \in ts[i := (p', is@is', \varnothing, sb, D, O, R)] \)

(\( \text{filter is-Write}_{sb} sb = [] \to sb = [] \))

by cases auto

from sim-loc [rule-format, OF i-bound] ts-i

obtain sb_h \( O_h R_h D_h \) \( acc_h \) where

\( ts_h-i : ts_h[i = (p, is, \varnothing, sb_h, D_h, O_h, R_h)] \) and

\( sb = \text{filter is-Write}_{sb} sb_h \) and

\( sb\text{-empty: filter is-Write}_{sb} sb_h = [] \to sb_h = [] \)

by (auto simp add: Let-def)

from lts-eq i-bound have i-bound': \( i < \text{length } ts_h \)

by simp

let \( ?ts_h-i' = (p', is @ is', \varnothing, sb_h @ \text{[Prog}_{sb} p p' is'], D_h, O_h, R_h) \)

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 \(= []\))
case False
have \(ts[i := (p', is@is', \emptyset, sb, D, O, R)] \sim_h ts'_h\)
apply (rule sim-history-config.intros)
using lts-eq
apply simp
using sim-loc i-bound i-bound' sb 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 \(?ts'_h, m, S_h\)
\(\Rightarrow_{shh} (ts'_h[i := (p', is@is', \emptyset, [], D_h, O_h, R_h), m, S_h])\)
by (simp add: empty)

with step have \((ts'_h, m, S_h) \Rightarrow_{shh}^* \)
\((ts'_h[i := (p', is@is', \emptyset, [], D_h, O_h, R_h), m, S_h])\)
by force

moreover
have \(ts[i := (p', is@is', \emptyset, sb, D, O, R)] \sim_h ts_h[i := (p', is@is', \emptyset, [], D_h, O_h, R_h)]\)
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 \(?\)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
\(ts': ts' = ts[i := (p, is, \emptyset, sb', D, O', R')]\) and
i-bound: \(i < \text{length ts}\) and
\(ts': i = (p, is, \emptyset, sb, D, O, R)\) and
sb-step: \((m, sb, O, R, S) \rightarrow_w (m', sb', O', R', S')\)
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 \(ts[i := (p, is, \emptyset, filter is-Write_{sb} sb, D', O', R') \land\)
by cases auto

from sim-loc [rule-format, OF i-bound] ts-i
obtain sb_h O_h D_h acq_h where
tsb-i: ts_h[i] = (p, is, θ, sb_h, D_h, O_h, R_h) and
sb: sb = filter is-Write sb_h and
sb-empty: filter is-Write sb_h = [] → sb_h = []
by (auto simp add: Let-def)

from lts-eq i-bound have i-bound': i < length ts_h
by simp

from flush-simulates-filter-writes [OF sb-step sb_h O_h D_h acq_h]
obtain sb_h' O_h' R_h' S_h' where
flush': (m, sb_h, O_h, R_h, S_h) → t' (m', sb_h', O_h', R_h', S_h') and
sb': sb' = filter is-Write sb_h' and
sb'-empty: filter is-Write sb_h' = [] → sb_h' = []
by auto

from sb-step obtain volatile a sop v A L R W where sb=Write sb_h volatile a sop v A L R W
by cases force
from sbh-computation.store-buffer-steps [OF flush' i-bound' tsb_h-i]
have (ts_h, m, S_h) ⇒ sbh * (ts_h[i := (p, is, θ, sb_h, D_h, O_h, R_h)], m', S_h').
moreover
have ts[i := (p, is, θ, sb', D_h, O_h', R_h') ] ~_h
 ts_h[i := (p, is, θ, sb_h, 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 sb' sb'-empty
apply (auto simp add: Let-def nth-list-update)
done
ultimately show ?thesis
by (auto simp add: ts')
qed

theorem (in valid-program) concurrent-history-steps-simulates-store-buffer-steps:
assumes step-sb: (ts, m, S) ⇒ sb * (ts', m', S')
shows ∃ ts_h. S_h. ts ~_ h ts_h → ∃ ts_h'. S_h'. (ts_h, m, S_h) ⇒ sb_h * (ts_h', m', S_h') ∧ ts' ~_ h ts_h'
using step-sb
proof (induct rule: converse-rtranclp-induct-sbh-steps)
case refl thus ?case by auto
next

qed
case (step ts m S \( \rightarrow \) ts'' m'' S'')
have first: (ts,m,S) \( \rightarrow \) (ts'',m'',S'') by fact
have sim: ts \( \sim \) ts_b by fact
from concurrent-history-steps-simulates-store-buffer-step [OF first sim, of S_h]
obtain ts_h'' S_h'' where
  exec: (ts_h,m,S_h) \( \Rightarrow \) sbb (ts''',m''',S_h') and sim: ts'' \( \sim \) h ts_h''
  by auto
from step.hyps (3) [OF sim, of S_h'']
obtain ts_h' S_h' where exec-rest: (ts_h'',m'',S_h'') \( \Rightarrow \) sbb (ts_h',m',S_h') and sim': ts' \( \sim \) h ts_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_ab,m_ab,x) \( \Rightarrow \) sbb (ts_ab',m_ab',x')
assumes init: initial_ab ts_ab S_ab
assumes valid: valid ts_ab
assumes sim: (ts_ab,m_ab,S_ab) \( \sim \) (ts,m,S)
assumes safe: safe-reach-direct-safe-free-flowing (ts,m,S)
shows \( \exists \) ts_h' S_h' ts' m' S'.
  \( \forall \) ts_ab m_ab S_ab (ts_ab,m_ab,S_ab) \( \Rightarrow \) sbb (ts_h',m_ab',S_h')
  ts_ab \( \sim \) h ts_h' 
  (ts,m,S) \( \Rightarrow \) d (ts',m',S') 
  (ts_h',m_ab',S_h') \( \sim \) (ts',m',S')
proof –
from init interpret ini: initial_ab ts_ab S_ab .
from concurrent-history-steps-simulates-store-buffer-steps [OF exec-sb ini.history-refl, of S_ab]
obtain ts_h' S_h' where
  sbh: (ts_ab,m_ab,S_ab) \( \Rightarrow \) sbb (ts_h',m_ab',S_h') and
  sim-sbh: ts_ab' \( \sim \) h ts_h'
  by auto
from concurrent-direct-execution-simulates-store-buffer-history-execution [OF sbh init valid sim safe]
obtain ts' m' S' where
  d: (ts,m,S) \( \Rightarrow \) d (ts',m',S') and
  d-sim: (ts_h',m_ab',S_h') \( \sim \) (ts',m',S')
  by auto
with sbh sim-sbh show \( ? \) thesis by blast
qed

inductive sim-direct-config::
  \( \langle \text{'p', p store-buffer, 'dirty', 'owns', 'rels'} \rangle \text{ thread-config list} \Rightarrow \langle \text{'p,unit,bool,'owns', 'rels'} \rangle \text{ thread-config list} \Rightarrow \text{bool} \)
  \( (- \sim_d - [60,60]) 100) \)
where
\[\text{length } ts = \text{length } ts_d;\]
\[\forall i < \text{length } ts.
(\exists O' D' R'.
\let (p,\text{is}, 0, \text{sb}, D, O, R) = ts_d!i \in
\text{ts}!i=(p,\text{is}, 0, [], D',O',R'))\]
\[\Rightarrow\]
\[ts \sim_d ts_d\]

lemma empty-sb-sims:

assumes empty:
\[\forall i \text{ p is xs sb } D O R. i < \text{length ts}_{\text{sb}} \rightarrow \text{ts}_{\text{sb}}!i=(p,\text{is},xs,\text{sb},D,O,R)\rightarrow \text{sb}=[]\]

assumes sim-h: \(ts_{\text{sb}} \sim_h ts_h\)

assumes sim-d: \((ts_h,m_h,S_h) \sim (ts,m,S)\)

shows \(ts_{\text{sb}} \sim_d ts \land m_h=m \land \text{length ts}_{\text{sb}} = \text{length ts}\)

proof–

from sim-h empty
have empty':
\[\forall i \text{ p is xs sb } D O R. i < \text{length ts}_h \rightarrow \text{ts}_h!i=(p,\text{is},xs,\text{sb},D,O,R)\rightarrow \text{sb}=[]\]

apply (cases)
apply clarsimp
subgoal for i
apply (drule-tac x=i in spec)
apply (auto simp add: Let-def)
done

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 (drule-tac x=i in spec)
apply (auto simp add: Let-def)
done
done

qed

lemma empty-d-sims:

assumes sim: \(ts_{\text{sb}} \sim_d ts\)

shows \(\exists ts_h. ts_{\text{sb}} \sim_h ts_h \land (ts_h,m,S) \sim (ts,m,S)\)

proof –

from sim obtain
leq: length ts_{\text{sb}} = length ts and
∀ \ i < \ \text{length} \ ts_{sb}.
(\exists \ O' \ D' \ \mathcal{R}'.
 \let (p, is, sb, D, O, R) = tsi \ in
 ts_{sb}!!i=(p, is, \emptyset, \emptyset, [\mathcal{D}'], O', \mathcal{R}')
)\\
by cases auto
\text{define} \ ts_h \ where \ ts_h \equiv \ \text{map} (\lambda (p, is, \theta, sb, D, O, R). (p, is, \emptyset, \emptyset, D', O', \mathcal{R}'))
\text{ts have} \ ts_{sb} \sim h \ ts_h
\text{apply} (\text{rule sim-history-config}.\text{intros})
\text{using} \ \text{leq sim}
\text{apply} (\text{auto simp add: ts_h-def Let-def leq})
\text{done}
\text{moreover}
\text{have empty:}
\forall \ i \ p \ is \ xs \ sb \ DOR. \ i < \ \text{length} \ ts_h \rightarrow ts_h!!i=(p, is, xs, sb, D, O, R) \rightarrow sb=\emptyset
\text{apply} (\text{clarsimp simp add: ts_h-def Let-def})
\text{subgoal for} \ i
\text{apply} (\text{case-tac ts}!i)
\text{apply auto}
\text{done}
\text{done}

\text{have} \ (ts_h, m, S) \sim (ts, m, S)
\text{apply} (\text{rule sim-config-emptyI [OF empty]})
\text{apply} (\text{clarsimp simp add: ts_h-def})
\text{apply} (\text{clarsimp simp add: ts_h-def Let-def})
\text{subgoal for} \ i
\text{apply} (\text{case-tac ts}!i)
\text{apply auto}
\text{done}
\text{done}
\text{ultimately show} ?thesis \ by \ blast
\text{qed}

\text{theorem} (\text{in xvalid-program-progress}) \ \text{concurrent-direct-execution-simulates-store-buffer-execution-empty:}
\text{assumes exec-sb:} \ (ts_{sb}, m_{sb}, x) \Rightarrow_{sb}^* (ts_{sb}', m_{sb}', x')
\text{assumes init:} \ \text{initial}_{sb} \ ts_{sb} \ S_{sb}
\text{assumes valid:} \ \text{valid} \ ts_{sb}
\text{assumes empty:}
\forall \ i \ p \ is \ xs \ sb \ D \ O \ R. \ i < \ \text{length} \ ts_{sb}' \rightarrow ts_{sb}!i=(p, is, xs, sb, D, O, R) \rightarrow sb=\emptyset
\text{assumes sim:} \ (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)
\text{assumes safe:} \ \text{safe-reach-direct safe-free-flowing} \ (ts, m, S)
\text{shows} \ \exists ts' S'.
(\text{ts, m, S}) \Rightarrow_{d}^* (ts', m_{sb}', S') \land ts_{sb}' \sim_d ts'
\text{proof} –
\text{from} \ \text{concurrent-direct-execution-simulates-store-buffer-execution [OF exec-sb init valid sim safe]}
\text{obtain} \ ts_h' \ S'_h \ ts' \ m' S' \ where
(t_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbh}^* (ts_h', m_{sb}', S'_h) \ \text{and}

715
\textbf{sim-h: ts}_sb′ ∼ h \textbf{and}
\textbf{exec: (ts,m,}\mathcal{\mathit{S}}) \Rightarrow ^*_d (ts′,m′,\mathcal{\mathit{S}}′) \textbf{and}
\textbf{sim: (ts}_h′,m_{sb′},\mathcal{\mathit{S}}_h′) \sim (ts′,m′,\mathcal{\mathit{S}}′)

by auto
from empty-sb-sims [OF empty sim-h sim]
obtain ts}_sb′ ∼ d ts′ m_{sb′} = m′ length ts}_sb′ = length ts′
by auto
thus \textbf{?thesis}
using exec
by blast
qed

locale initial\_d = simple-ownership-distinct + read-only-unowned + unowned-shared +
fixes valid
assumes empty-is: \[i \textless \text{length ts}; ts!i=(p,is,}sb,\mathcal{\mathit{D}},\mathcal{\mathit{O}},\mathcal{\mathit{R}})\] \implies is=[]
assumes empty-rels: \[i \textless \text{length ts}; ts!i=(p,is,}sb,\mathcal{\mathit{D}},\mathcal{\mathit{O}},\mathcal{\mathit{R}})\] \implies R=Map.empty
assumes valid-init: valid (map (\lambda (p,is,}sb,\mathcal{\mathit{D}},\mathcal{\mathit{O}},\mathcal{\mathit{R}}). (p,is, \emptyset,\mathcal{\mathit{D}}, \mathcal{\mathit{O}},\mathcal{\mathit{R}})) ts)

locale empty-store-buffers =
fixes ts::(\textbf{p,}\textbf{p store-buffer,}bool,owns,rels) thread-config list
assumes empty-sb: \[i \textless \text{length ts}; ts!i=(p,is,}sb,\mathcal{\mathit{D}},\mathcal{\mathit{O}},\mathcal{\mathit{R}})\] \implies sb=[]

lemma initial-d-sb:
assumes init: initial\_d ts \mathcal{\mathit{S}} valid
shows initial\_sb (map (\lambda (p,is,}sb,\mathcal{\mathit{D}},\mathcal{\mathit{O}},\mathcal{\mathit{R}}). (p,is, \emptyset,\mathcal{\mathit{D}}, \mathcal{\mathit{O}},\mathcal{\mathit{R}})) ts) \mathcal{\mathit{S}}
(is initial\_sb ?map \mathcal{\mathit{S}})

proof –
from init interpret ini: initial\_d ts \mathcal{\mathit{S}} .
show \textbf{?thesis}
proof (intro-locales)
show simple-ownership-distinct \textbf{?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 assumption
apply assumption
apply auto
done
done
next
show read-only-unowned \mathcal{\mathit{S}} \textbf{?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)

716
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_{ab}-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
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 $\exists ts' S'$. $\Rightarrow_d^* (ts',m',S') \land ts_{sb}' \sim_d ts'$
proof –
  from empty'
have empty':
\[ \forall i \ p \is xs \sb \D \O \R. \ i < \text{length } \ts_{\sb} \rightarrow \ts_{\sb}!i=(p,\is,\xs,\sb,\D,\O,\R) \rightarrow \sb=[] \]
by (auto simp add: empty-store-buffers-def)

define \( \ts_{\h} \) where  
\( \ts_{\h} \equiv \text{map } (\lambda (p,\is,\theta,\sb,\D,O,R). (p,\is,\theta,\[],[]::\text{a memref list},\D,O,R)) \)
\( \ts \)
from initial-d-sb [OF init]
have init-h:initial \( \ts_{\h} \) \( \S \)
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: \( \forall i < \text{length } \ts_{\sb} \).  
\( (\exists \O', \D', \R'. \ (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: \( (\ts_{\h},m,\S) \Rightarrow^{*}_{\sb} (\ts_{\h}',m',\S_{\h}') \) and
sim-h': \( \ts_{\sb}' \sim_{\h} \ts_{\h}' \)
by auto
moreover
have empty':
\( \forall i \ p \is xs \sb \D \O \R. \ i < \text{length } \ts_{\h} \rightarrow \ts_{\h}!i=(p,\is,\xs,\sb,\D,\O,\R) \rightarrow \sb=[] \)
apply (clarsimp simp add: ts\_h-def Let-def)
subgoal for i
apply (case-tac ts!i)
apply auto
done
done

have sim': (\( \ts_{\h},m,\S \)) ~ (ts,m,\( \S \))
apply (rule sim-config-emptyI [OF empty])
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
from concurrent-direct-execution-simulates-store-buffer-history-execution [OF steps-h init-h valid-h sim', safe]

obtain \( t's\) \( m'' S'' \) where steps: \( (t's, m, S) \Rightarrow_d^* (t's, m'', S') \)
and sim': \( (t's, m', S, \cdot) \sim (t's', m'', S') \)
by blast

from empty-sb-sims [OF empty sim-h sim'] 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 (\( \lambda\)(p, is, \( \emptyset\), sb, D, O, R)) ts)

theorem (in xvalid-program-progress) store-buffer-execution-result-sequential-consistent' :
assumes exec-sb: \( (t's_{sb}, m, x) \Rightarrow_{sb}^* (t's_{sb}', m', x') \)
assumes empty': empty-store-buffers t's_{sb}'
assumes sim: \( t's_{sb} \sim_d t's \)
assumes init: initial\( v \) t's S valid
assumes safe: safe-reach-virtual safe-free-flowing (t's,m,S)
shows \( \exists t's', S'. \)
\( (t's, m, S) \Rightarrow_v^* (t's', m', S') \land t's_{sb}' \sim_d t's' \)
proof
- define \( t's_d \) where \( t's_d == (map (\( \lambda\)(p, is, \( \emptyset\), sb, D, O, R), (p, is, \( \emptyset\),[],D, O, Map.empty)) ts) \)
have rem-ts: remove-rels t's_d = t's
  apply (rule nth-equalityI)
  apply (simp add: t's_d-def remove-rels-def)
  apply (clarsimp simp add: t's_d-def remove-rels-def)
  apply (clarsimp)
  apply (clarsimp)
done
from sim
have sim': t's_{sb} \sim_d t's_d
  apply cases
  apply (rule sim-direct-config.intros)
  apply (auto simp add: t's_d-def)
done

have init': initial\( v \) t's_d S valid
proof (intro-locales)
  from init have simple-ownership-distinct t's

719
by (simp add: initial\textsubscript{\lambda}-def)

then

show simple-ownership-distinct ts\textsubscript{d}
  apply (clarsimp simp add: ts\textsubscript{d}-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\textsubscript{\lambda}-def)
then show read-only-unowned S ts\textsubscript{d}
  apply (clarsimp simp add: ts\textsubscript{d}-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\textsubscript{\lambda}-def)
then

show unowned-shared S ts\textsubscript{d}
  apply (clarsimp simp add: ts\textsubscript{d}-def unowned-shared-def)
  apply force
  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\textsubscript{\lambda}-axioms ts valid
  by (simp add: initial\textsubscript{\lambda}-def)

then

show initial\textsubscript{\lambda}-axioms ts\textsubscript{d} valid
  apply (clarsimp simp add: ts\textsubscript{d}-def initial\textsubscript{\lambda}-axioms-def initial\textsubscript{\lambda}-axioms-def eq)
  apply (rule conjI)
  apply clarsimp
    subgoal for i
    apply (case-tac ts!i)
    apply force
    done
apply clarsimp
subgoal for i
apply (case-tac ts!i)
apply force
done
done
qed

{(fix tsd' m' S'
assume exec: (tsd, m, S) ⇒* (tsd', m', S')
have safe-free-flowing (tsd', m', S')
proof –
from virtual-simulates-direct-steps [OF exec]
have exec-v: (ts, m, S) ⇒* (remove-rels tsd', m', S')
by (simp add: rem-ts)
have eq: map (owned ◦ (λ(p, is, ð, sb, D, O, R). (p, is, ð, ()), D, O, ()))))
tsd' = map owned tsd'
by auto
from exec-v safe
have safe-free-flowing (remove-rels tsd', m', S')
by (auto simp add: safe-reach-def)
then 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 (tsd, m, S)
by (simp add: safe-reach-def)

from store-buffer-execution-result-sequential-consistent [OF exec-sb empty' sim' init' safe']
obtain tsd' S' where
exec-d: (tsd, m, S) ⇒* (tsd', m', S') and sim-d: tsdb' ∼d tsd'
by blast

from virtual-simulates-direct-steps [OF exec-d]
have (ts, m, S) ⇒* (remove-rels tsd', m', S')
by (simp add: rem-ts)
moreover
from sim-d
have tsdb' ∼d remove-rels tsd'
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 xvalid-program) concurrent-direct-steps-simulates-store-buffer-step:

assumes step-sb: (ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sb} (ts_{sb}', m_{sb}', S_{sb}')

assumes sim-h: ts_{sb} \sim_{h} ts_{sb}'

assumes sim: (ts_{sbh}, m_{sb}, S_{sbh}) \sim (ts, m, S)

assumes valid-own: valid-ownership S_{sbh} ts_{sbh}

assumes valid-sb-reads: valid-reads m_{sb} ts_{sbh}

assumes valid-history: valid-history program-step ts_{sbh}

assumes valid-sharing: valid-sharing 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, S)

shows \exists ts_{sbh}', S_{sbh}'.

\forall (ts_{sbh}, m_{sb}, S_{sbh}) \Rightarrow_{sbh} (ts_{sbh}', m_{sb}', S_{sbh}') \land ts_{sbh}' \sim_{h} ts_{sbh}' \land

valid-ownership S_{sbh} ts_{sbh}' \land valid-reads m_{sb} ts_{sbh}' \land

valid-history program-step ts_{sbh}' \land

valid-sharing S_{sbh} ts_{sbh}' \land tmps-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}'

proof –

from concurrent-history-steps-simulates-store-buffer-step [OF step-sb sim-h]

obtain ts_{sbh}', S_{sbh}' where

steps-h: (ts_{sbh}, m_{sb}, S_{sbh}) \Rightarrow_{sbh} (ts_{sbh}', m_{sb}', S_{sbh}') and

sim-h': ts_{sbh}' \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-history valid-sharing tmps-distinct valid-sops valid-dd load-tmps-fresh enough-flushs valid-program-history valid sim safe-reach]

obtain m' ts' S' where

\forall (ts, m, S) \Rightarrow_{d} (ts, m', S') (ts_{sbh}', m_{sb}', S_{sbh}') \sim (ts', m', S') \land

valid-ownership S_{sbh} ts_{sbh}' \land valid-reads m_{sb} ts_{sbh}' \land valid-history program-step ts_{sbh}'

valid-sharing S_{sbh} ts_{sbh}' \land tmps-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}' \land valid ts_{sbh}'

by blast

ultimately

show ?thesis

by blast

qed
lemma conj-commI: \( P \land Q \implies Q \land P \)
by simp

lemma def-to-eq: \( P = Q \implies P \equiv Q \)
by simp

class xvalid-program
begin

definition invariant \( ts \ S m \equiv \)
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 \( \equiv \) valid-ownership

definition sharing-inv \( \equiv \) valid-sharing

definition temporaries-inv ts \( \equiv \) tmps-distinct ts \( \land \) load-tmps-fresh ts

definition history-inv ts m \( \equiv \) valid-history program-step ts \( \land \) valid-program-history ts \( \land \)
valid-reads m ts

definition data-dependency-inv ts \( \equiv \) valid-data-dependency ts \( \land \) load-tmps-fresh ts \( \land \)
valid-sops ts

definition barrier-inv \( \equiv \) enough-flushs

lemma invariant-grouped-def: invariant \( ts \ S m \equiv \)
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 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
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
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 tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns 
| CAS expr expr expr tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns 
| Seq stmt stmt 
| Cond expr stmt stmt 
| While expr stmt 
| SGhost tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns tmps ⇒ 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 e 1 e 2) = used-tmps e 1 + used-tmps e 2

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 e 1 e 2) = issue-expr t e 1 @ issue-expr (t + (used-tmps e 1)) e 2

primrec eval-expr:: tmp ⇒ expr ⇒ sop — calculate result 
where
eval-expr t (Const v) = ({},λθ. v) 
| eval-expr t (Mem volatile a) = ({t},λθ. the (θ t)) 
| eval-expr t (Tmp sop) = sop

end
— trick to enforce sop to be sensible in the current context, without
having to include wellformedness constraints
\[ \text{eval-expr } t \ (\text{Unop } f \ e) = (\text{let } (D, f e) = \text{eval-expr } t \ e \text{ in } (D, \lambda \theta. f (f e \theta))) \]
\[ \text{eval-expr } t \ (\text{Binop } f e_1 e_2) = (\text{let } (D_1, f_1) = \text{eval-expr } t \ e_1; \]
\[ (D_2, f_2) = \text{eval-expr } (t + (\text{used-tmps } e_1)) \ e_2 \]
\[ \text{in } (D_1 \cup D_2, \lambda \theta. f (f_1 \theta) (f_2 \theta))) \]

**primrec** valid-sops-expr:: nat ⇒ expr ⇒ bool
**where**
\[ \text{valid-sops-expr } t \ (\text{Const } v) = \text{True} \]
\[ \text{valid-sops-expr } t \ (\text{Mem } \text{volatile } a) = \text{True} \]
\[ \text{valid-sops-expr } t \ (\text{Tmp } \text{sop}) = ((\forall t' \in \text{fst sop. } t' < t) \land \text{valid-sop } \text{sop}) \]
\[ \text{valid-sops-expr } t \ (\text{Unop } f \ e) = \text{valid-sops-expr } t \ e \]
\[ \text{valid-sops-expr } t \ (\text{Binop } f e_1 e_2) = (\text{valid-sops-expr } t \ e_1 \land \text{valid-sops-expr } t \ e_2) \]

**primrec** valid-sops-stmt:: nat ⇒ stmt ⇒ bool
**where**
\[ \text{valid-sops-stmt } t \ \text{Skip} = \text{True} \]
\[ \text{valid-sops-stmt } t \ (\text{Assign } \text{volatile } a \ e \ A \ L \ R \ W) = (\text{valid-sops-expr } t \ a \land \text{valid-sops-expr } t \ e) \]
\[ \text{valid-sops-stmt } t \ (\text{CAS } a \ c_e \ s_e \ A \ L \ R \ W) = (\text{valid-sops-expr } t \ a \land \text{valid-sops-expr } t \ c_e \land \text{valid-sops-expr } t \ s_e) \]
\[ \text{valid-sops-stmt } t \ (\text{Seq } s_1 s_2) = (\text{valid-sops-stmt } t \ s_1 \land \text{valid-sops-stmt } t \ s_2) \]
\[ \text{valid-sops-stmt } t \ (\text{Cond } e s_1 s_2) = (\text{valid-sops-expr } t \ e \land \text{valid-sops-stmt } t \ s_1 \land \text{valid-sops-stmt } t \ s_2) \]
\[ \text{valid-sops-stmt } t \ (\text{While } e \ s) = (\text{valid-sops-expr } t \ e \land \text{valid-sops-stmt } t \ s) \]
\[ \text{valid-sops-stmt } t \ (\text{SGhost } A \ L \ R \ W) = \text{True} \]
\[ \text{valid-sops-stmt } t \ \text{SFence} = \text{True} \]

**type-synonym** stmt-config = stmt × nat
**consts** isTrue:: val ⇒ bool

**inductive** stmt-step:: tmps ⇒ stmt-config ⇒ stmt-config × instrs ⇒ bool
\[ (- \vdash - \rightarrow_s - [60,60,60] \ 100) \]
**for** \( \delta \)
**where**
\[ AssignAddr: \]
\[ \forall \text{sop. } a \neq \text{Tmp } \text{sop} \rightarrow \]
\[ \delta \vdash (\text{Assign } \text{volatile } a \ e \ A \ L \ R \ W, t) \rightarrow_s \]
\[ ((\text{Assign } \text{volatile } (\text{Tmp } (\text{eval-expr } t \ a)) \ e \ A \ L \ R \ W, t + \text{used-tmps } a), \text{issue-expr } t \ a) \]

| Assign: \[ \text{D} \subseteq \text{dom } \delta \rightarrow \]
\[ \delta \vdash (\text{Assign } \text{volatile } (\text{Tmp } (\text{D}, a)) \ e \ A \ L \ R \ W, t) \rightarrow_s \]
\[ ((\text{Skip}, t + \text{used-tmps } e), \]
\[ \text{issue-expr } t \ e \delta | \text{Write } \text{volatile } (a \ \delta) \ (\text{eval-expr } t \ e) \ (A \ \delta) \ (L \ \delta) \ (R \ \delta) \ (W \ \delta))] \]

725
\[
\text{CASAddr:} \\
\forall \text{sop. } a \neq \text{Tmp sop} \implies \\
\emptyset \vdash (\text{CAS } a c e s e A L R W, t) \rightarrow_s \\
((\text{CAS (Tmp (eval-expr t a)) } c e s e A L R W, t + \text{used-tmps a}), \text{issue-expr t a})
\]

\[
\text{CASComp:} \\
\forall \text{sop. } c e \neq \text{Tmp sop} \implies \\
\emptyset \vdash (\text{CAS (Tmp (D a, a)) } c e s e A L R W, t) \rightarrow_s \\
((\text{CAS (Tmp (D a, a)) } (\text{Tmp (eval-expr t c e)) } s e A L R W, t + \text{used-tmps c e}), \\
\text{issue-expr t c e})
\]

\[
\text{CAS:} \\
[D_a \subseteq \text{dom } \emptyset; D_c \subseteq \text{dom } \emptyset; \text{eval-expr t s e } = (D, f)] \\
\implies \\
\emptyset \vdash (\text{CAS (Tmp (D a, a)) } (\text{Tmp (D c, c)) } s e A L R W, t) \rightarrow_s \\
((\text{Skip, Suc (t + used-tmps s e)}), \text{issue-expr t s e @} \\
[\text{RMW (a } \emptyset) (t + \text{used-tmps s e}) (D, f) (\lambda \emptyset. \text{the } (\emptyset (t + \text{used-tmps s e})) = c \emptyset) (\lambda v_2. v_1) \\
(A \emptyset) (L \emptyset) (R \emptyset) (W \emptyset)])
\]

\[
\text{Seq:} \\
\emptyset \vdash (s_1, t) \rightarrow_s ((s_1', t'), \text{is}) \\
\implies \\
\emptyset \vdash (\text{Seq } s_1 s_2, t) \rightarrow_s ((\text{Seq } s_1' s_2, t'), \text{is})
\]

\[
\text{SeqSkip:} \\
\emptyset \vdash (\text{Seq Skip } s_2, t) \rightarrow_s ((s_2, t), [])
\]

\[
\text{Cond:} \\
\forall \text{sop. } e \neq \text{Tmp sop} \implies \\
\emptyset \vdash (\text{Cond } e s_1 s_2, t) \rightarrow_s \\
((\text{Cond (Tmp (eval-expr t e)) } s_1 s_2, t + \text{used-tmps e}), \text{issue-expr t e})
\]

\[
\text{CondTrue:} \\
[D \subseteq \text{dom } \emptyset; \text{isTrue } (e \emptyset)] \\
\implies \\
\emptyset \vdash (\text{Cond (Tmp (D e)) } s_1 s_2, t) \rightarrow_s ((s_1, t), [])
\]

\[
\text{CondFalse:} \\
[D \subseteq \text{dom } \emptyset; \neg \text{isTrue } (e \emptyset)] \\
\implies \\
\emptyset \vdash (\text{Cond (Tmp (D e)) } s_1 s_2, t) \rightarrow_s ((s_2, t), [])
\]

\[
\text{While:} \\
\emptyset \vdash (\text{While } e s, t) \rightarrow_s
\]

((Cond e (Seq s (While e s)) Skip, t),[])

| SGhost:
\[ \emptyset \vdash (SGhost A L R W, t) \rightarrow_{s} ((Skip, t),[Ghost (A \emptyset) (L \emptyset) (R \emptyset) (W \emptyset)]) \]

| SFence:
\[ \emptyset \vdash (SFence, t) \rightarrow_{s} ((Skip, t),[Fence]) \]

**inductive-cases stmt-step-cases [cases set]:**
\[ \emptyset \vdash (Skip, t) \rightarrow_{s} c \]
\[ \emptyset \vdash (Assign volatile a e A L R W, t) \rightarrow_{s} c \]
\[ \emptyset \vdash (Seq s_{1} s_{2}, t) \rightarrow_{s} c \]
\[ \emptyset \vdash (Cond e s_{1} s_{2}, t) \rightarrow_{s} c \]
\[ \emptyset \vdash (While e s, t) \rightarrow_{s} c \]
\[ \emptyset \vdash (SGhost A L R W, t) \rightarrow_{s} c \]
\[ \emptyset \vdash (SFence, t) \rightarrow_{s} c \]

**Lemma valid-sops-expr-mono:** \( \forall t t'. \) valid-sops-expr t e \( \Rightarrow \) t \( \leq \) t' \( \Rightarrow \) valid-sops-expr t' e

**Proof** (induct e)

**Case** (Unop f e)
then obtain valid-sops-expr t e by simp
from Unop.hyps [OF this]
have vs: valid-sop (eval-expr t e) by simp
obtain D g where eval-e: eval-expr t e = (D,g) by (cases eval-expr t e)
interpret valid-sop (D,g)
using vs eval-e by simp

**Show** ?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-tmps e₁)]
valid-sops-expr-mono [OF v2, of (t + used-tmps e₁)]

obtain vs1: valid-sop (eval-expr t e₁) and vs2: valid-sop (eval-expr (t + used-tmps 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-tmps e₁) e₂ = (D₂,g₂)
  by (cases eval-expr (t + used-tmps 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
}

728
qed

note lem=this

show ?case
  apply (clarsimp simp add: Let-def valid-sop-def eval-e1 eval-e2)
  apply (rule lem)
  by (auto simp add: valid-sop-def)

qed (auto simp add: valid-sop-def)

lemma valid-sops-expr-eval-expr-in-range:
\[ \forall t. \text{valid-sops-expr } t \ e \implies \forall t' \in \text{fst (eval-expr } t \ e), \ t' < t + \text{used-tmps } e \]

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: \( \theta \vdash (s, t) \rightarrow_{\delta} ((s', t'), \text{is}) \)
shows t \leq 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: \( \theta \vdash (s, t) \rightarrow_{\delta} ((s', t'), \text{is}) \)
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 \implies D \subseteq 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,0,SB,S,m) \rightarrow_{sbh} (is',O',D',0',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

730
lemma sbh-step-preserves-read-tmps-bound:
assumes step: (is, θ, sb, m, D, O, S) →sbh (is', θ', sb', m', D', O', S')
assumes less-is: ∀i ∈ load-tmps is. i < n
assumes less-sb: ∀i ∈ read-tmps sb. i < n
shows ∀i ∈ 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, θ, sb, m, D, O, S) →sbh (is', θ', sb', m', D', O', S')
assumes less-dom: ∀i ∈ dom θ. i < n
assumes less-is: ∀i ∈ load-tmps is. i < n
shows ∀i ∈ dom θ'. 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) →f (m', sb', O')
assumes less-sb: ∀i ∈ read-tmps sb. i < n
shows ∀i ∈ 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) →f (m', sb', O')
assumes less-sb: ∀i ∈ ∪ (fst ' write-sops sb). i < t
shows ∀i ∈ ∪ (fst ' write-sops sb'). i < t
using step less-sb
by cases (auto simp add: read-tmps-append)

lemma issue-expr-load-tmps-range':
∀t. load-tmps (issue-expr t e) = {i. t ≤ i ∧ i < t + used-tmps e}
apply (induct e)
apply (force simp add: load-tmps-append)+
done

lemma issue-expr-load-tmps-range:
∀t. ∀i ∈ load-tmps (issue-expr t e). t ≤ i ∧ i < t + (used-tmps e)
by (auto simp add: issue-expr-load-tmps-range')

lemma stmt-step-load-tmps-range':
assumes step: θ ⊢ (s, t) →s ((s', t'), is)
shows load-tmps is = {i. t ≤ i ∧ 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

731
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: $\forall t. \text{distinct-load-tmps (issue-expr } t \text{ e)}$
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: $t + \text{used-tmps } e \notin \text{load-tmps (issue-expr } t \text{ e)}$
proof
from issue-expr-load-tmps-range [rule-format, of $t+\text{used-tmps } e$]
show ?thesis
by auto
qed

lemma stmt-step-distinct-load-tmps:
assumes step: $\emptyset \vdash (s, t) \rightarrow_{s} ((s', t'),is)$
shows $\text{distinct-load-tmps is}$
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]: $\forall t. \text{store-sops (issue-expr } t \text{ e) = \{\}}$
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: $\theta \vdash (s,t) \rightarrow_{a} ((s',t'),is')$
  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' $\land$ (load-tmps is' $\cap$ load-tmps is = {}) $\land$
    (load-tmps is' $\cap$ read-tmps sb) = {}
proof –
  load-tmps-le read-tmps-le
  show ?thesis
    by force
qed

733
lemma data-dependency-consistent-intrs-issue-expr:
\( \forall t \text{ T. data-dependency-consistent-intrs T (issue-expr t e)} \)
apply (induct e)
apply (auto simp add: data-dependency-consistent-intrs-append
dest!: issue-expr-load-tmps-range [rule-format])
done

lemma dom-eval-expr:
\( \forall t. \neg \exists x \in \text{fst (eval-expr t e)} \Rightarrow x \in \{i. i < t\} \cup \text{load-tmps (issue-expr t e)} \)
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-s_1: s_1 \neq \text{Cond e s_1 s_2}
by (induct s_1) auto

lemma Cond-not-s_2: s_2 \neq \text{Cond e s_1 s_2}
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

734
lemma prog-step-progress:
assumes step: \( \vdash (s, t) \rightarrow_s ((s', t'), is) \)
shows \( (s', t') \neq (s, t) \lor is \neq [] \)
using step
proof (induct x== (s, t) y== ((s', t'), is)) arbitrary: s t s t' is rule: stmt-step.induct)
case (AssignAddr a - - - - - - - t) thus ?case
  by (cases eval-expr t a) auto
next
case Assign thus ?case by auto
next
case (CASAddr a - - - - - - - t) thus ?case by (cases eval-expr t a) auto
next
case (CASComp c_e - - - - - - - t) thus ?case by (cases eval-expr t c_e) auto
next
case CAS thus ?case by auto
next
case (Cond e - - - t) thus ?case by (cases eval-expr t e) auto
next
case CondTrue thus ?case using Cond-not-s_1 by auto
next
case CondFalse thus ?case using Cond-not-s_2 by auto
next
case Seq thus ?case by force
next
case SeqSkip thus ?case using Seq-not-s_2 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 stmt-step-data-dependency-consistent-instrs:
assumes step: \( \vdash (s, t) \rightarrow_s ((s', t'), is) \)
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'), is)) arbitrary: s t s t' is T rule: stmt-step.induct)
case AssignAddr
  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 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: \( \theta \vdash (s, t) \rightarrow (s', t', i, s') \)
  assumes store-sops-le: \( \forall i \in \bigcup (\text{fst}^t \text{ store-sops is}). \ i < t \)
  assumes write-sops-le: \( \forall i \in \bigcup (\text{fst}^t \text{ write-sops sb}). \ i < t \)
assumes valid: valid-sops-stmt t s
shows data-dependency-consistent-instrs \{i. i < t\} is' ∧
load-tmps is' ∪ (fst i store-sops is) = {} ∧
load-tmps is' ∪ (fst i write-sops sb) = {}

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' ∩ 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
  case CASAddr
  thus ?case by auto
next
  case CASComp
  thus ?case by auto
next
  case CAS

737
thus \( ?\text{case} \)
   by (fastforce simp add: store-sops-append dest: valid-sops-expr-valid-sop)
next
case Seq
case \( ?\text{case} \)
   by (force intro: valid-sops-stmt-mono)
next
case SeqSkip
case \( ?\text{case} \)
   by simp
next
case Cond
case \( ?\text{case} \)
   by auto
next
case CondTrue
case \( ?\text{case} \)
   by auto
next
case CondFalse
case \( ?\text{case} \)
   by auto
next
case While
case \( ?\text{case} \)
   by auto
next
case SGhost
case \( ?\text{case} \)
   by auto
next
case SFence
case \( ?\text{case} \)
   by auto
qed

primrec prog-configs:: \( 'a \ \text{memref list} \Rightarrow 'a \ \text{set} \)
where
prog-configs [] = {}
|prog-configs (x#xs) = (case x of
   
   Prog\_sb p p' is \Rightarrow \{p,p'\} \cup \text{prog-configs} xs
   |
   _ \Rightarrow \text{prog-configs} xs)

lemma prog-configs-append:
\( \forall ys. \ \text{prog-configs} (xs@ys) = \text{prog-configs} xs \cup \text{prog-configs} ys \)
   by (induct xs) (auto split: memref.splits)

lemma prog-configs-in1: Prog\_sb p_1 p_2 is \in set xs \Rightarrow p_1 \in \text{prog-configs} xs
   by (induct xs) (auto split: memref.splits)

lemma prog-configs-in2: Prog\_sb p_1 p_2 is \in set xs \Rightarrow p_2 \in \text{prog-configs} xs
   by (induct xs) (auto split: memref.splits)

lemma prog-configs-mono:
\( \forall ys. \ \text{set} xs \subseteq \text{set} ys \Rightarrow \text{prog-configs} xs \subseteq \text{prog-configs} ys \)
   by (induct xs) (auto split: memref.splits simp add: prog-configs-append prog-configs-in1 prog-configs-in2)

locale separated-tmps =
  fixes ts
  assumes valid-sops-stmt: \[ i < \text{length} ts; tsi = ((s,t),is,\emptyset,\emptyset,D,O) \]
  \Rightarrow valid-sops-stmt t s
  assumes valid-sops-stmt-sb: \[ i < \text{length} ts; tsi = ((s,t),is,\emptyset,\emptyset,D,O); (s',t') \in \text{prog-configs} sb \]

738
\[ \Rightarrow \text{valid-sops-stmt } t' s' \]

**assumes** load-tmps-le: \[ i < \text{length } ts; ts!i = ((s, t), is, \theta, sb, D, O) \]
\[ \Rightarrow \forall i \in \text{load-tmps is. } i < t \]

**assumes** read-tmps-le: \[ i < \text{length } ts; ts!i = ((s, t), is, \theta, sb, D, O) \]
\[ \Rightarrow \forall i \in \text{read-tmps sb. } i < t \]

**assumes** store-sops-le: \[ i < \text{length } ts; ts!i = ((s, t), is, \theta, sb, D, O) \]
\[ \Rightarrow \forall i \in \bigcup (\text{fst ' store-sops is}). i < t \]

**assumes** write-sops-le: \[ i < \text{length } ts; ts!i = ((s, t), is, \theta, sb, D, O) \]
\[ \Rightarrow \forall i \in \bigcup (\text{fst ' write-sops sb}). i < t \]

**assumes** tmps-le: \[ i < \text{length } ts; ts!i = ((s, t), is, \theta, sb, D, O) \]
\[ \Rightarrow \forall i \in \bigcup (\text{fst ' tmps sb}). i < t \]

**assumes** dom \( \theta \cup \text{load-tmps is} = \{ i. i < t \} \)

**lemma (in separated-tmps)**

**tmps-le':**

**assumes** i-bound: \( i < \text{length } ts \)

**assumes** ts-i: \( ts!i = ((s, t), is, \theta, sb, D, O) \)

**shows** \( \forall i \in \text{dom } \theta. i < t \)

**using** tmps-le [OF i-bound ts-i] **by** auto

**lemma (in separated-tmps)** separated-tmps-nth-update:

\[ i < \text{length } ts; \text{valid-sops-stmt } t s; \forall (s', t') \in \text{prog-configs sb. valid-sops-stmt } t' s'; \]
\[ \forall i \in \text{load-tmps is. } i < t; \forall i \in \text{read-tmps sb. } i < t; \]
\[ \forall i \in \bigcup (\text{fst ' store-sops is}). i < t; \forall i \in \bigcup (\text{fst ' write-sops sb}). i < t; \text{dom } \theta \cup \text{load-tmps is} = \{ i. i < t \} \]
\[ \Rightarrow \]

separated-tmps (\( ts[i:=((s, t), is, \theta, sb, D, O)] \))

**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: \( \forall ys. \text{Prog}_{sb} p p' is \in \text{set } xs \Rightarrow \text{hd-prog } q (xs @ ys) = \text{hd-prog } q xs \)

**by** (induct xs) (auto split: memref.splits)

**lemma** hd-prog-app-in-eq: \( \forall ys. \text{Prog}_{sb} p p' is \in \text{set } xs \Rightarrow \text{hd-prog } q xs = \text{hd-prog } q x s \)

**by** (induct xs) (auto split: memref.splits)

739
lemma hd-prog-app-notin-first: \( \forall p \ p' \ is. \ Progsb p p' \notin set xs \Rightarrow \text{hd-prog } q (xs \at \ ys) = \text{hd-prog } q\ ys \)
by (induct xs) (auto split: memref.splits)

lemma union-eq-subsetD: A \cup B = C \Rightarrow A \cup B \subseteq C \land C \subseteq A \cup B 
by auto

lemma prog-step-preserved-separated-tmps:
  assumes i-bound: \( i < \text{length ts} \)
  assumes ts-i: ts!i = (p, is, \theta, sb, D, O)
  assumes prog-step: \( \theta \vdash p \rightarrow s (p', \text{is}') \)
  assumes sep: separated-tmps ts
  shows separated-tmps (ts[i := ((s', t'), is@is', \theta, sb@[Progsb p p' is']], D, O)])
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 = prog-step [simplified p p']
  interpret separated-tmps ts by fact
  have separated-tmps (ts[i := ((s', t'), is@is', \theta, sb@[Progsb (s, t) (s', t') 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 \( \forall i \in \text{read-tmps } (\text{sb@[Progsb (s, t) (s', t') is']}) . \ i < t' \)
  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 \( \forall i \in \bigcup (\text{fst } \text{store-sops } (\text{is@is'})) . \ i < t' \)
  by (fastforce simp add: store-sops-append)
next
  show \( \forall i \in \bigcup (\text{fst } \text{write-sops } (\text{sb@[Progsb (s, t) (s', t') is']})) . \ i < t' \)
  by (fastforce simp add: write-sops-append)
next
  from tmps-le [OF i-bound ts-i]
  have dom \( \theta \cup \text{load-tmps is} = \{ i . \ i < t \} \)
  by simp
  show dom \( \theta \cup \text{load-tmps is@is'} = \{ i . \ i < t' \} \)
  apply (clarsimp simp add: load-tmps-append)
  apply rule
  apply (drule union-eq-subsetD)
  apply fastforce

740
apply clarsimp
apply (case-tac t \leq 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 \forall (s', t') \in prog-configs (sb \oplus [\text{Prog}_{sb} (s, t) (s', t') is']).
valid-sops-stmt t' s'

proof -

{ fix s_1 t_1
assume cfgs: (s_1, t_1) \in prog-configs (sb \oplus [\text{Prog}_{sb} (s, t) (s', t') is'])
have valid-sops-stmt t_1 s_1
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

lemma flush-step-sb-subset:
assumes step: (m, sb, O) \rightarrow_f (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
lemma flush-step-preserves-separated-tmps:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb, D, O, R)
assumes flush-step: (m, sb, O, R, S) → (m', sb', O', R', S')
assumes sep: 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 load-tmps-le [OF i-bound ts-i]
show ∀i∈load-tmps is. i < t.
next
show ∀i∈read-tmps sb'. i < t.
next
from store-sops-le [OF i-bound ts-i]
show ∀i∈∪(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 ∀i∈∪(fst ' write-sops sb'). i < t.
next
from tmps-le [OF i-bound ts-i]
show dom θ ∪ 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 ∀(s', t')∈prog-configs sb'. valid-sops-stmt t' s'
  by (auto dest!: prog-configs-mono)
qed
then
show ?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)

done

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:=((s, t), 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:=(p, 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
    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 ∀i∈read-tmps sb'. i < t.
next
    from sbh-step-preserves-store-sops-bound [OF memop-step store-sops-le [OF i-bound ts-i]]
    show ∀i∈(fst ' store-sops is'). i < t.

743
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 \( \forall i \in \bigcup (\text{fst } \text{write-sops sb}') \). \( i < t \).
next
from sbh-step-preserves-tmps-bound' [OF memop-step] 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
show \( \forall (s', t') \in \text{prog-configs sb'} \). valid-sops-stmt t' s'
  by auto
qed
then show \(?\)thesis
  by (simp add: p)
qed

definition
valid-pimp ts \equiv \text{separated-tmps ts}

lemma prog-step-preserves-valid:
  assumes i-bound: \( i < \text{length ts} \)
  assumes ts-i: ts!i = (p, is, \theta, \text{sb::stmt-config store-buffer}, D, O, R)
  assumes prog-step: \( \theta \vdash p \rightarrow_s (p', is') \)
  assumes valid: valid-pimp ts
  shows valid-pimp (ts [i:=(p', is', \theta, \text{sb}])]
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, \theta, \text{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, \theta, 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: \( i < \text{length ts} \)
  assumes ts-i: ts!i = (p, is, \theta, \text{sb::stmt-config store-buffer}, D, O, R)
  assumes memop-step: (is, \theta, sb, m, D, O, R, S) \rightarrow_{sbh}
  (is', \theta', sb', m', D', O', R', S')
  assumes valid: valid-pimp ts
  shows valid-pimp (ts [i:=(p, is', \theta', 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 λ(s,t). valid-sops-stmt t s
valid-pimp

proof
  fix θ p p′ is'
  assume step: θ ⊢ p →ₜ (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' ≠ p ∨ is' ≠ []
    by (simp add: p p')
next
  fix θ p p′ is'
  assume step: θ ⊢ p →ₜ (p', is')
    and valid-stmt: (λ(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 (λ(s, t). valid-sops-stmt t s) p' by (simp add: p')
next
  fix i ts p is O R D θ sb
  assume i-bound: i < length ts
    and ts-i: ts ! i = (p, is, θ, 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 (λ(s, t). valid-sops-stmt t s) p
  by (auto simp add: p)
next
  fix i ts p is O R D θ sb
  assume i-bound: i < length ts
    and ts-i: ts ! i = (p, is, θ, 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 hd-prog-prog-configs [of p 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). valid-sops-stmt t s) (hd-prog p sb)
  by (auto simp add: p)

next

fix i ts p is O R D \emptyset sb p' is'

assume i-bound: i < 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 distinct-load-tmps is' ∧
  load-tmps is' ∩ load-tmps is = {} ∧
  load-tmps is' ∩ 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)

  read-tmps-le [OF i-bound ts-i]]

show ?thesis .

qed

next

fix i ts p is O R D \emptyset sb p' is'

assume i-bound: i < 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 data-dependency-consistent-instrs (dom \emptyset \cup load-tmps is) is' ∧
  load-tmps is' ∩ \bigcup (fst ' store-sops is) = {} ∧
  load-tmps is' ∩ \bigcup (fst ' write-sops 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 sbh-valid-data-dependency-prog-step [OF step store-sops-le [OF i-bound ts-i]
  read-sops-le [OF i-bound ts-i]]

show ?thesis .

qed

show ?thesis by auto
qed

next
fix i ts p is O R D \varnothing sb p' is'
assume i-bound: i < length ts
and ts-i: ts ! i = (p, is, \varnothing, sb, D, O,R)
and step: \varnothing \vdash p \rightarrow_s (p', is')
and valid: valid-pimp ts
show load-tmps is' \cap \text{dom} \varnothing = {}

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 sbh-load-tmps-fresh-prog-step [OF step tmps-le' [OF i-bound ts-i]]
show ?thesis .
qed

next
fix \varnothing p p' is
assume step: \varnothing \vdash p \rightarrow_s (p', is)
and valid: (\lambda(s, t). 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': 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 \varnothing sb p' is'
assume i-bound: i < length ts
and ts-i: ts ! i = (p, is, \varnothing, sb:stmt-config store-buffer, D, O,R)
and step: \varnothing \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' \circ is', \varnothing, sb [@ [Prog sb p p' is']], D, O,R)]) .

next
fix i ts p is O R D \varnothing sb \mathcal{S} m m' sb' O' R' S'
assume i-bound: i < length ts
and ts-i: ts ! i = (p, is, \varnothing, sb:stmt-config store-buffer, D, O,R)
and step: (m, sb, O, R,S) \rightarrow_f (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, θ, sb', D', O', R')])

next
fix i ts p is \( \mathcal{O} \mathcal{R} \mathcal{D} \mathcal{S} \) m is' \( \mathcal{O}' \mathcal{R}' \mathcal{D}' \mathcal{S}' \) m'
assume i-bound: i < length ts
  and ts-i: ts ! i = (p, is, θ, sb::stmt-config store-buffer, D, O, R)
  and step: (is, θ, sb, m, D, O, R, S) \( \rightarrow_{sbh} \)
  (is', θ', sb', m', D', O', R', S')
  and valid: valid-pimp ts
from sbh-step-preserves-valid [OF i-bound ts-i step valid]
show valid-pimp (ts[i := (p, is', θ', 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
  \( \lambda \ p \ p' \ is \ sb . sb @ [\text{Prog}_{sb} \ p \ p' \ is] \).

abbreviation direct-pimp-step::
  (stmt-config,unit,bool,owns,rels/shared) global-config \( \Rightarrow \)
  (stmt-config,unit,bool,owns,rels/shared) global-config \( \Rightarrow \) bool
  (- \( \Rightarrow_{dp} \) \([60,60]\ 100)\)
where
  c \( \Rightarrow_{dp} \) d \( \equiv \) 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 \( \Rightarrow \) bool
  (- \( \Rightarrow_{dp}^* \) \([60,60]\ 100)\)
where
  direct-pimp-steps == direct-pimp-step\(^\ast\ast\)

Execution examples

lemma Assign-Const-ex:

(\( \lambda \theta . \ A \)) (\( \lambda \theta . \ L \)) (\( \lambda \theta . \ R \)) (\( \lambda \theta . \ W \)) \( t \),\( . \),\( . \) \( \mathcal{O} \mathcal{R} \mathcal{D} \mathcal{O} \mathcal{R} \mathcal{S} \) \( \Rightarrow_{dp}^* \)
(\( \lambda \theta . \ True \cup A \cup R, Map.empty \)) \( m(a := c) \),\( S \oplus W \ R \oplus A \ L \)
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Program \( \text{where} \ i=0)\)
apply simp
apply simp
apply (rule Assign)
apply simp
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Memop \( \text{where} \ i=0)\)
apply simp
apply simp

748
apply simp
apply (rule direct-memop-step.WriteVolatile)
apply simp
done

lemma

\[
\begin{align*}
&\text{\texttt{((Assign True (Tmp \{\},\lambda \theta. a)) (Binop (+) (Mem True x) (Mem True y)) (\lambda \theta. A) (\lambda \theta. L) (\lambda \theta. R) (\lambda \theta. W),t),[\theta,(\emptyset,D,O,R)],m,S)} \\
&\Rightarrow_{dp}^* \\
&\text{\texttt{((Skip,t + 2),[\theta(t := m x, t + 1 := m y),(),True,O \cup A - R,Map.empty]),m(a := m x + m y),S \oplus_{W} R \ominus_{A} L)}}
\end{align*}
\]
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)
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

apply simp

lemma

assumes isTrue: isTrue c
shows

\[
\begin{align*}
&\text{\texttt{((Cond (Const c) (Assign True (Tmp \{\},\lambda \theta. a)) (Const c) (\lambda \theta. A) (\lambda \theta. L) (\lambda \theta. R) (\lambda \theta. W)) Skip,t),[\theta,(\emptyset,D,O,R)],m,S)} \\
&\Rightarrow_{dp}^* \\
&\text{\texttt{((Skip,t),[\theta,(\emptyset,D,O,R)],m(a := c),S \oplus_{W} R \ominus_{A} L)}}
\end{align*}
\]
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct.Program [where i=0])
apply simp

749
apply simp
apply (rule Cond)
apply simp
apply simp
apply (rule converse-rtranclp-into-rtranclp)
apply (rule direct. Program [where i=0])
apply simp
apply simp
apply (rule CondTrue)
apply simp
apply simp
apply simp
apply simp
apply (rule Assign-Const-ex)
done

end

References