A Reduction Theorem for Store Buffers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

2 Preliminaries

The formalization presented in this 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., nat ⇒ nat is a total function on natural numbers. A function update is f(y := v) = (λx. if x = y then v 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., [x], and the additional element ⊥. A base value can be projected with the function the, which is defined by the sole equation the [x] = x. Since HOL is a total logic the term the ⊥ is still a well-defined yet un(der)specified value. Partial functions are usually represented by the type ’a ⇒ ’b option, abbreviated as ’a → ’b. They are commonly used as maps. We denote the domain of map m by dom m. A map update is written as m(a → v). We can restrict the domain of a map m to a set A by m|A.

The syntax and the operations for lists are similar to functional programming languages like ML or Haskell. The empty list is [], with x # xs the element x is ‘consed’ to the list xs. With xs @ ys list ys is appended to list xs. With the term 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 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 ∪ B, membership, i.e., x ∈ 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-volatilley, 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 (read-write)</td>
<td>shared (read-only)</td>
</tr>
<tr>
<td>unowned vR, vW</td>
<td>vR, R unreachable</td>
</tr>
<tr>
<td>owned vR, vW, R</td>
<td>unreachable</td>
</tr>
<tr>
<td>owned by other vR</td>
<td>unreachable</td>
</tr>
</tbody>
</table>

(v)olatile, (R)ead, (W)rite all reads have to be clean
4 Formalization

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

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

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

- **Read volatile** \( a t \) for reading from address \( a \) to temporary \( t \), where the Boolean \( volatile \) determines whether the access is volatile or not.
- **Write volatile** \( a sop A L R W \) to write the result of evaluating the store operation \( sop \) at address \( a \). A store operation is a pair \((D, f)\), with the domain \( D \) and the function \( f \). The function \( f \) takes temporaries \( \theta \) as a parameter, which maps a temporary to a value. The subset of temporaries that is considered by function \( f \) is specified by the domain \( D \). We consider store operations as valid when they only depend on their domain:

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

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

- **FENCE**, a memory fence that flushes the store buffer.

- **GHOST** A L R W for ownership transfer.

### 4.1 Store buffer machine

For the store buffer machine the configuration of a single thread is a tuple \((p, is, \varnothing, sb)\) consisting of the program state \( p \), a memory instruction list \( is \), the map of temporaries \( \varnothing \) 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.

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

Fig. 2: Global transitions of store buffer machine

A transition selects a thread \( ts[i] = (p, is, \varnothing, sb) \) 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 \( \varnothing \vdash p \rightarrow_p (p', is') \) takes the temporaries \( \varnothing \) 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 \((is, \varnothing, sb, m) \xrightarrow{sb} (is', \varnothing', sb', m')\) of a machine with store buffer may only fill its store buffer with new writes.

In a store buffer step \((m, sb) \xrightarrow{sb} (m', sb')\) 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 \texttt{WRITE}_{sb} volatile a sop v for an outstanding write (keeping the volatile flag), where operation sop evaluated to value v.

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

Fig. 3: Store buffer transition

\[v = (\text{case buffered-val } sb a \text{ of } \bot \Rightarrow m a | [v'] \Rightarrow v')\]

\[(\text{Read volatile } a t \# is, \vartheta, sb, m) \vdash_{m} (is, \vartheta(t \rightarrow v), sb, m)\]

\[sb' = sb \oplus [\text{Write}_{vb} \text{ volatile } a (D, f) (f \vartheta) A L R W]\]

\[(\text{Write volatile } a (D, f) A L R W \# is, \vartheta, sb, m) \vdash_{m} (is, \vartheta, sb', m)\]

\[\neg \text{ cond } (\vartheta(t \rightarrow m a)) \quad \vartheta' = \vartheta(t \rightarrow m a)\]

\[(\text{RMW } a t (D, f) \text{ cond ret } A L R W \# is, \vartheta, [\], m) \vdash_{m} (is, \vartheta', [\], m)\]

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

\[(\text{RMW } a t (D, f) \text{ cond ret } A L R W \# is, \vartheta, [\], m) \vdash_{m} (is, \vartheta', [\], m')\]

\[(\text{Fence } \# is, \vartheta, [\], m) \vdash_{m} (is, \vartheta, [\], m)\]

\[(\text{Ghost } A L R W \# is, \vartheta, sb, m) \vdash_{m} (is, \vartheta, sb, m)\]

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 \((p, is, \vartheta, D, \mathcal{O})\), with a dirty flag \(D\) indicating whether there may be an outstanding volatile write in the store buffer and the set of owned addresses \(\mathcal{O}\). 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 \(O s, i \vdash (is, \vartheta, m, D, \mathcal{O}, S)\) \(\surd\) in Figure 7, where \(O s\) is the list of ownership sets obtained from the thread list \(ts\) and \(i\) is the index of the current
\[
i < |ts| \quad ts[|i|] = (p, is, \emptyset, D, O) \quad \theta \vdash p \rightarrow p' (p', is') \\
(ts, m, S) \Rightarrow (ts[i := (p', is \oplus is', \emptyset, D, O)], m, S)
\]

\[
i < |ts| \quad ts[|i|] = (p, is, \emptyset, D, O) \quad(is, \emptyset, m, D, O, S) \rightarrow_m (is', \emptyset', m', D', O', S') \\
(ts, m, S) \Rightarrow (ts[i := (p, is', \emptyset', D')], m', S')
\]

Fig. 5: Global transitions of virtual machine

---

(Read volatile a t # is, \emptyset, x, m, ghst) \rightarrow_m (is, \emptyset(t \rightarrow 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 - 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')

\rightarrow \text{cond} (\emptyset(t \rightarrow 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, x, m, ghst')

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

m' = m(a := f (\emptyset(t \rightarrow m a))) \quad ghst = (D, O, S) \quad ghst' = (\text{False}, O \cup A - 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 - 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
thread. Safety of all reachable states of the virtual machine ensures that the programming discipline is obeyed by the program and is our formal prerequisite for the simulation theorem. It is left as a proof obligation to be discharged by means of a proper program logic for sequentially consistent executions. In the following we elaborate on the rules of

\[
\begin{align*}
    & a \in \mathcal{O} \lor a \in \text{read-only } \mathcal{S} \lor \text{volatile } \land a \in \text{dom } \mathcal{S} \quad \text{volatile } \rightarrow \lnot \mathcal{D} \\
    & \mathcal{O}_s, \vdash \text{ (Read volatile } a \not\in \emptyset, \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
    & a \in \mathcal{O} \quad a \not\in \text{dom } \mathcal{S} \\
    & \mathcal{O}_s, \vdash \text{ (Write False } a (D, f) \ A \ L \ R \ W \ # \ is, \ \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
    & \forall j < |\mathcal{O}_s|, i \neq j \rightarrow a \notin \mathcal{O}_s[j] \quad a \notin \text{read-only } \mathcal{S} \\
    & \mathcal{O}_s, \vdash \text{ (Write True } a (D, f) \ A \ L \ R \ W \ # \ is, \ \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
    & \lnot \text{cond } (\emptyset(t \rightarrow m a)) \quad a \in \text{dom } \mathcal{S} \cup \mathcal{O} \\
    & \mathcal{O}_s, \vdash \text{ (RMW a t (D, f) cond ret A L R W # is, } \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
    & \forall j < |\mathcal{O}_s|, i \neq j \rightarrow A \cap \mathcal{O}_s[j] = \emptyset \quad A \subseteq \mathcal{O} \cup \text{dom } \mathcal{S} \\
    & \mathcal{O}_s, \vdash \text{ (RMW a t (D, f) cond ret A L R W # is, } \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
    & \forall j < |\mathcal{O}_s|, i \neq j \rightarrow A \cap \mathcal{O}_s[j] = \emptyset \quad A \cap R = \emptyset \\
    & A \subseteq \text{dom } \mathcal{S} \cup \mathcal{O} \\
    & \mathcal{O}_s, \vdash \text{ (Fence # is, } \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
    & \mathcal{O}_s, \vdash \text{ (Ghost } A \ L \ R \ W \ # \ is, \ \emptyset, m, \mathcal{D}, \mathcal{O}, \mathcal{S}) \sqrt{12} \\
\end{align*}
\]

Fig. 7: Safe configurations of a virtual machine

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

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

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

The read-modify-write instruction first adds the current value at address \(a\) to temporary \(t\) and then checks the store condition \(\text{cond}\) on the temporaries. If it fails this read is
the final result of the operation. Otherwise the store is performed. The resulting value of the temporary $t$ is specified by the function $ret$ which considers both the old and new value as input. As the read-modify-write instruction is an interleaved 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 $Ghost A L R W$ allows to transfer ownership when no write is involved i.e., when merely reading from memory. It has the same safety requirements as the corresponding parts in the write instructions.

### 4.3 Reduction

The reduction theorem we aim at reduces a computation of a machine with store buffers to a sequential consistent computation of the virtual machine. We formulate this as a simulation theorem which states that a computation of the store buffer machine $(ts_{sb}, m) \xrightarrow{\emptyset \ast} (ts_{sb}^\prime, m^\prime)$ can be simulated by a computation of the virtual machine $(ts, m, S) \xrightarrow{\emptyset \ast} (ts^\prime, m^\prime, S^\prime)$. The main theorem only considers computations that start in an initial configuration where all store buffers are empty and end in a configuration where all store buffers are empty again. A configuration of the store buffer machine is obtained from a virtual configuration by removing all ghost components and assuming empty store buffers. This coupling relation between the thread configurations is written as $ts_{sb} \sim ts$. Moreover, the precondition $initial_v ts S$ 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 $safe-reach (ts, m, S)$ we ensure conformance to the programming discipline by assuming that all reachable configuration in the virtual machine are safe (according to the rules in Figure 7).

**Theorem 1 (Reduction).**

$$(ts_{sb}, m) \xrightarrow{\emptyset \ast} (ts_{sb}^\prime, m^\prime) \land empty\text{-}store\text{-}buffers ts_{sb}^\prime \land ts_{sb} \sim ts \land initial_v ts S \land$$

$$safe-reach (ts, m, S) \rightarrow$$

$$\exists ts^\prime S^\prime (ts, m, S) \xrightarrow{\emptyset \ast} (ts^\prime, m^\prime, S^\prime) \land ts_{sb}^\prime \sim ts^\prime$$

This theorem captures our intuition that every result that can be obtained from a computation 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 intermediate 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 configuration of a machine with store buffers and the virtual machine without store buffers. It
allows us to simulate every computation step of the store buffer machine by a sequence of steps (potentially empty) on the virtual machine. This transformation is essentially a sequentialization of the trace of the store buffer machine. When a thread of the store buffer machine executes a non-volatile operation, it only accesses memory which is not modified by any other thread (it is either owned or read-only). Although a non-volatile store is buffered, we can immediately execute it on the virtual machine, as there is no competing store of another thread. However, with volatile writes we have to be careful, since concurrent threads may also compete with some volatile write to the same address.

At the moment the volatile write enters the store buffer we do not yet know when it will be issued to memory and how it is ordered relatively to other outstanding writes of other threads. We therefore have to suspend the write on the virtual machine from the moment it enters the store buffer to the moment it is issued to memory. For volatile reads our programming discipline guarantees that there is no volatile write in the store buffer by flushing the store buffer if necessary. So there are at most some outstanding non-volatile writes in the store buffer, which are already executed on the virtual machine, as described before. One simple coupling relation one may think of is to suspend the whole store buffer as not yet executed instructions of the virtual machine. However, consider the following scenario. A thread is reading from a volatile address. It can still have non-volatile writes in its store buffer. Hence the read would be suspended in the virtual machine, and other writes to the address (e.g. interlocked or volatile writes of another thread) could invalidate the value. Altogether this suggests the following refined coupling relation: the state of the virtual machine is obtained from the state of the store buffer machine, by executing each store buffer until we reach the first volatile write. The remaining store buffer entries are suspended as instructions. As we only execute non volatile writes the order in which we execute the store buffers should be irrelevant. This coupling relation allows a volatile read to be simulated immediately on the virtual machine as it happens on the store buffer machine.

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

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

5.1 Intermediate models 

We begin with the virtual machine with delayed releases, for which the memory transitions 
\((is, \varnothing, m, D, O, R, S) \xrightarrow{m} (is', \varnothing', m', D', O', R', S')\) 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:

\[
\text{aug (dom } S) \rightarrow R = 
\]
\[(\text{Read volatile} \ a \ t \ # \ is, \ \emptyset, \ m, \ \text{ghst}) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (is, \ \emptyset(t \rightarrow m \ a), \ m, \ \text{ghst})\]

\[(\text{Write False} \ a \ (D, f) \ A \ L \ R \ W \ # \ is, \ \emptyset, \ m, \ \text{ghst}) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (is, \ \emptyset, \ m(a := f \ \emptyset), \ \text{ghst})\]

\[\text{ghst} = (D, O, R, S) \quad \text{ghst}’ = (\text{True}, O \cup A - R, \emptyset \oplus_W R \ominus_A L)\]

\[(\text{Write True} \ a \ (D, f) \ A \ L \ R \ W \ # \ is, \ \emptyset, \ m, \ \text{ghst}) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (is, \ \emptyset, \ m(a := f \ \emptyset), \ \text{ghst}’)\]

\[\neg \text{cond} (\emptyset(t \rightarrow m \ a)) \quad \text{ghst} = (D, O, R, S) \quad \text{ghst}’ = (\text{False}, O \cup A - R, \emptyset \oplus_W R \ominus_A L)\]

\[(\text{RWM} \ a \ (D, f) \ \text{cond ret} \ A \ L \ R \ W \ # \ is, \ \emptyset, \ m, \ \text{ghst}) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (is, \ \emptyset’, \ m’, \ \text{ghst}’)\]

\[(\text{Fence} \ # \ is, \ \emptyset, \ m, D, O, R, S) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (is, \ \emptyset, \ m, \ False, O, \emptyset, S)\]

\[\text{ghst} = (D, O, R, S) \quad \text{ghst}’ = (D, O \cup A - R, \text{aug} \ (\text{dom} \ S) \ R R, \ S \oplus_W R \ominus_A L)\]

\[
\begin{align*}
\text{cond} (\emptyset(t \rightarrow m \ a)) & \quad \emptyset’ = \emptyset(t \rightarrow (m a) (f (\emptyset(t \rightarrow m \ a)))) \\
& \quad m’ = m(a := f (\emptyset(t \rightarrow m \ a))) \\
& \quad \text{ghst} = (D, O, R, S) \\
& \quad \text{ghst}’ = (\text{False}, O \cup A - R, \emptyset \oplus_W R \ominus_A L) \\
\end{align*}
\]

\[(\text{Ghost} \ A \ L \ R \ W \ # \ is, \ \emptyset, \ m, \ \text{ghst}) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (is, \ \emptyset, \ m, \ \text{ghst}’)\]

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

\[(\lambda a. \text{if } a \in R \text{ then case } R \ a \text{ of } \bot \Rightarrow [a \in \text{dom} \ S] \mid [s] \Rightarrow [s \land a \in \text{dom} \ S]\]

\[\text{else } \emptyset \ a)\]

If an address is freshly released \((a \in R \text{ and } R \ a = \bot)\) the flag is set according to \text{dom} \ S. Otherwise the flag becomes \([\text{False}]\) in case the released address is currently unshared.

Note that with this definition \(R \ a = [\text{False}]\) stays stable upon every new release and we do not loose information about a release of an unshared address.

The global transition \((ts, m, s) \xrightarrow{\text{\text{\texttt{}}} \text{\texttt{}}} (ts’, m’, s’)\) 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 \(\mathcal{O}s, R, s, i |- (is, \ \emptyset, \ m, D, O, S) \) \(\quad\) 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 \((a \notin \text{dom } Rs[i])\); reading from an address is only allowed if it is not released by another thread while it was unshared \((Rs[i], a \neq [\text{False}]\).

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

\begin{itemize}
  \item \text{READ}_{sb} \ \text{volatile} \ a \ t \ v, \ \text{recording a corresponding read from address} \ a \ \text{which loaded the value} \ v \ \text{to temporary} \ t, \ \text{or a}
  \item \text{WRITE}_{sb} \ \text{volatile} \ a \ \text{so} \ v \ \text{for an outstanding write, where operation} \ \text{so} \ \text{evaluated to value} \ v, \ \text{or of the form}
  \item \text{PROC}_{sb} \ p \ p’ \ is’, \ \text{recording a program transition from} \ p \ \text{to} \ p’ \ \text{which issued instructions} \ is’, \ \text{or of the form}
  \item \text{GHOST}_{sb} \ A \ L \ R \ W, \ \text{recording a corresponding ghost operation}.
\end{itemize}

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

\[O_{s}, R_{s}, i \vdash (\text{Read volatile } a t \# \text{ is, } \varnothing, m, D, O, S) \checkmark\]
\[a \in O \land a \notin \text{dom } S \land \forall j \in |O|, i \neq j \rightarrow a \notin \text{dom } R_{S_{ij}}\]
\[O_{s}, R_{s}, i \vdash (\text{Write False } a (D, f) A L R W \# \text{ is, } \varnothing, m, D, O, S) \checkmark\]
\[\forall j \in |O|, i \neq j \rightarrow a \notin O_{S_{ij}} \cup \text{dom } R_{S_{ij}}\]
\[a \notin \text{read-only } S \land \forall j \in |O|, i \neq j \rightarrow A \cap (O_{S_{ij}} \cup \text{dom } R_{S_{ij}}) = \emptyset\]
\[A \subseteq \text{dom } S \cup O \land L \subseteq A \land R \subseteq O \land A \cap R = \emptyset\]
\[O_{s}, R_{s}, i \vdash (\text{Write True } a (D, f) A L R W \# \text{ is, } \varnothing, m, D, O, S) \checkmark\]

\[\neg \text{cond } (\varnothing(t \rightarrow m a)) \land a \in \text{dom } S \cup O \land \forall j \in |O|, i \neq j \rightarrow R_{S_{ij}} \neq a \neq |\text{False}|
\]
\[O_{s}, R_{s}, i \vdash (\text{RMW } a t (D, f) \text{ cond ret } A L R W \# \text{ is, } \varnothing, m, D, O, S) \checkmark\]
\[\text{cond } (\varnothing(t \rightarrow m a)) \land a \in \text{dom } S \cup O \land \forall j \in |O|, i \neq j \rightarrow a \notin O_{S_{ij}} \cup \text{dom } R_{S_{ij}}\]
\[a \notin \text{read-only } S \land \forall j \in |O|, i \neq j \rightarrow A \cap (O_{S_{ij}} \cup \text{dom } R_{S_{ij}}) = \emptyset\]
\[A \subseteq \text{dom } S \cup O \land L \subseteq A \land R \subseteq O \land A \cap R = \emptyset\]
\[O_{s}, R_{s}, i \vdash (\text{Fence } A L R W \# \text{ is, } \varnothing, m, D, O, S) \checkmark\]
\[O_{s}, R_{s}, i \vdash ([], \varnothing, m, D, O, S) \checkmark\]

\[\text{Fig. 9: Safe configurations of a virtual machine (delayed-releases)}\]

\[\text{(m, Write}_{ab} \text{ False } a \text{ sop } v A L R W \# sb, O, R, S) \rightarrow_{abh} \text{ (m}(a := v)\text{, sb, O, R, S)}\]
\[O' = O \cup A - R \quad S' = S \oplus_{W} R \oplus_{A} L\]
\[(m, \text{Write}_{ab} \text{ True } a \text{ sop } v A L R W \# sb, O, R, S) \rightarrow_{abh} \text{ (m}(a := v)\text{, sb, O', empty, S')}\]
\[\text{(m, Read}_{ab} \text{ volatile } a t v \# sb, O, R, S) \rightarrow_{abh} \text{ (m, sb, O, R, S)}\]
\[\text{(m, Prog}_{ab} \text{ p p' is } \# \text{ sb, O, R, S)} \rightarrow_{abh} \text{ (m, sb, O, R, S)}\]
\[O' = O \cup A - R \quad R' = \text{aug (dom S)} R R \quad S' = S \oplus_{W} R \oplus_{A} L\]
\[(m, \text{Ghost}_{ab} A L R W \# sb, O, R, S) \rightarrow_{abh} \text{ (m, sb, O', R', S')}\]

\[\text{Fig. 10: Store buffer transitions with history}\]
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 \((ts_{sbh}, m, S) \xrightarrow{sb} (ts_{sbh}', m', S')\)

\[
\begin{align*}
\nu &= \text{(case buffered-val sb a of } \bot \Rightarrow m a \mid [\nu'] \Rightarrow \nu') \quad sb' = sb \oplus \text{[Read}_{sb} \text{volatile a t v]} \\
(\text{Read volatile a t # is, } \vartheta, sb, m, ghst) &\xrightarrow{sb} (is, \vartheta(t \mapsto v), sb', m, ghst) \\
(\text{Write False a (D, f) A L R W }\# \text{ is, } \vartheta, sb, m, ghst) &\xrightarrow{sb} (is, \vartheta, sb', m, ghst) \\
(\text{Write True a (D, f) A L R W }\# \text{ is, } \vartheta, sb, m, ghst) &\xrightarrow{sb} (is, \vartheta, sb', m, ghst') \\
\neg \text{ cond } (\vartheta(t \mapsto m a)) &\xrightarrow{gghst = (D, O, R, S, \epsilon)} (\text{False, O, empty, S}) \\
(\text{RMW a t (D, f) cond ret A L R W }\# \text{ is, } \vartheta, [], m, ghst) &\xrightarrow{m} (\text{False, } O \cup A \rightarrow R, \text{ empty, } S \oplus W R \cup A L) \\
(\text{Fence # is, } \vartheta, [], m, D, O, R, S) &\xrightarrow{ghst} (is, \vartheta, [], m, \text{False, O, empty, S}) \\
(\text{Ghost A L R W }\# \text{ is, } \vartheta, sb, m, G) &\xrightarrow{ghst} (is, \vartheta, sb \oplus \text{[Ghost}_{sb} A L R W}, m, G) \\
\end{align*}
\]

Fig. 11: Memory transitions of store buffer machine with history are analogous to the rules in Figure 2 replacing the memory transitions and store buffer transitions accordingly.

5.2 Coupling relation

After this introduction of the immediate models we can proceed to the details of the coupling relation, which relates configurations of the store buffer machine with history and the virtual machine with delayed releases. Remember the basic idea of the coupling relation: the state of the virtual machine is obtained from the state of the store buffer machine, by executing each store buffer until we reach the first volatile write. The remaining store buffer entries are suspended as instructions. The instructions now also include the history entries for reads, program steps and ghost operations. The suspended reads are not yet visible in the temporaries of the virtual machine. Similar the ownership effects (and program steps) of the suspended operations are not yet visible in the virtual machine. The coupling relation between the store buffer machine and the virtual machine is illustrated in Figure 12. The thread issue instructions to the store buffers from the right and the instructions emerge from the left 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:

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

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 $execs$ and the suspended ones by $suspends$. The function $instrs$ converts them back to instructions, which are a prefix of the instructions of the virtual
Machine. We collect the additional instructions which were issued by program instructions but still recorded in the remainder of the store buffer with function \( \text{prog-instrs} \). These instructions have already made their way to the instructions of the store buffer machine but still recorded in the remainder of the store buffer with function \( \text{prog-instrs} \). These instructions are formalized as \( \text{instrs \ suspends} @ i_{sbh} = \text{is} @ \text{prog-instrs \ suspends} \), where \( \text{is} \) are the instructions of the virtual machine. The program state of the virtual machine is either the same as in the store buffer machine or the first program state recorded in the suspended part of the store buffer. This state is selected by \( \text{hd-prog} \). The temporaries of the virtual machine are obtained by removing the suspended reads from \( \theta \). The memory is obtained by executing all store buffers until the first volatile write is hit, excluding it. Thereby only non-volatile writes are executed, which are all thread local, and hence could be executed in any order with the same result on the memory. Function \( \text{exec-all-until-volatile-write} \) 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 \( \text{share-all-until-volatile-write} \). For the local ownership set \( \text{O}_{sbh} \) the auxiliary function \( \text{acquire} \) calculates the outstanding effect of the already simulated parts of the store buffer. Analogously \( \text{release} \) calculates the effect for the released addresses \( \text{R}_{sbh} \).

### 5.3 Simulation

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

**Theorem 2 (Simulation).**

\[
\begin{align*}
(t_{sbh}, m_{sbh}, S_{sbh}) \xrightarrow{h} (t'_{sbh}, m'_{sbh}, S'_{sbh}) \wedge (t_{sbh}, m_{sbh}, S_{sbh}) & \sim (ts, m, S) \wedge \text{safe-reach-delayed} (ts, m, S) \wedge \text{invariant} \ t_{sbh} \ S_{sbh} \ m_{sbh} \\
\text{invariant} \ t_{sbh} \ S_{sbh} \ m_{sbh} & \wedge \\
(\exists ts' S' m') (ts, m, S) \xrightarrow{d} (ts', m', S') \wedge (t_{sbh}, m_{sbh}, S_{sbh}) & \sim (ts', m', S')
\end{align*}
\]

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

\[
\text{invariant} \ t_{sbh} S_{sbh} m_{sbh} \equiv \text{ownership-inv} S_{sbh} t_{sbh} \wedge \text{sharing-inv} S_{sbh} t_{sbh} \wedge
\]

![Fig. 13: Coupling relation](image-url)
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 $\theta$.

Data dependency. Data dependency means that store operations may only depend on previous read operations. For every thread we have: (i) Every operation $(D, f)$ in a write instruction or a store buffer write is valid according to $\text{valid-sop} (D, f)$, i.e., function $f$ only depends on domain $D$. (ii) For every suffix of the instructions of the form $\text{WRITE volatile } a (D, f) A L R W \# is$ 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 $\text{WRITE}_{sb} \text{ volatile } a (D, f) v A L R W$ the recorded value $v$ coincides with $f \theta$, and domain $D$ is subset of $\text{dom } \theta$ 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 $\text{PROG}_{sb} p_1 p_2 is' \# sb'$, either $p_1 = p$ in case there is no preceding program node in the buffer or it corresponds to the last program state recorded there. Moreover, the program transition $\theta'((- \text{read-tmps } sb')) \vdash p_1 \rightarrow_p (p_2, is')$ 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 exemplarically 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 exemplarically 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 \( i \): 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 wheter 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 $c_{fg''} = c_{fg}$. 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 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)](image)

We refer to $x$ both for the write operation itself and to characterize the point in time in the computation of the virtual machine where the write was executed. At the point $x$ the write was safe according to rules in Figure 9 for non-volatile writes. So it was owned by thread $j$ and unshared. This knowledge about the safety of write $x$ is preserved in the invariants, namely (Ownership.$i$) and (Sharing.$i$). Additionally from invariant (Sharing.$v$) we know that address $a$ was not read-only at point $x$. Now we combine this information with the safety of the read of thread $i$ in the current configuration $c_{fg}$: 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 thread $j$, 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 $R_{S_{j}} = \{ \text{False} \}$. 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 $c_{fg''}$ 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 an 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 destination 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 \( (ts, m, S) \xrightarrow{\nu} (ts', m', S') \) 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 \( (ts, m, S) \xrightarrow{\nu} (ts', m', S') \)\(^5\).

\(^5\) Here we are sloppy with \( ts \); strictly we would have to distinguish the thread configurations without the \( \mathcal{R} \) component form the ones with the \( \mathcal{R} \) 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 safe-reach \((ts, m, S)\) provided by the preconditions of Theorem 1 to the required safe-reach-delayed \((ts, m, S)\). This argument is more involved and we only give a short sketch here. The other direction is trivial as every single case for delayed releases (cf. Figure 9) immediately implies the corresponding case for free flowing releases (cf. Figure 7).

First keep in mind that the predicates ensure that all reachable configurations starting from \((ts, m, S)\) are safe, according to the rules for free flowing releases or delayed releases respectively. We show the theorem by contraposition and start with a computation which reaches a configuration \(c\) that is unsafe according to the rules for delayed releases and want to show that there has to be a (potentially other) computation (starting from the same initial state) that leads to an unsafe configuration \(c'\) accroding 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 simultananeously 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) \texttt{Const} \(v\), a constant value, (ii) \texttt{Mem volatile} \(a\), a (volatile) memory lookup at address \(a\), (iii) \texttt{Tmp} \(sop\), reading from the temporaries with a operation \(sop\) which is an intermediate expression occurring in the transition rules for statements,
(iv) **Unop** $f \ e$, a unary operation where $f$ is a unary function on values, and finally
(v) **Binop** $f \ e_1 \ e_2$, a binary operation where $f$ is a binary function on values.

A statement $s$ is either (i) **Skip**, the empty statement, (ii) **Assign volatile** $a \ e \ A \ L \ R \ W$, a (volatile) assignment of expression $e$ to address expression $a$, (iii) **CAS** $a \ e_1 \ s_e \ A \ L \ R \ W$, 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 $used-tmps \ e$ calculates

\[
\begin{align*}
\text{issue-expr} \ (t \ \text{const} \ v) &= \ [] \\
\text{issue-expr} \ (t \ \text{mem volatile} \ a) &= [\text{read volatile} \ a \ t] \\
\text{issue-expr} \ (t \ \text{tmp} \ (D, f)) &= \ [] \\
\text{issue-expr} \ (t \ \text{unop} \ f \ e) &= \text{issue-expr} \ t \ e \\
\text{issue-expr} \ (t \ \text{binop} \ f \ e_1 \ e_2) &= \text{issue-expr} \ t \ e_1 \ @ \text{issue-expr} \ (t + used-tmps \ e_1) \ e_2 \\
\text{eval-expr} \ (t \ \text{const} \ v) &= (\emptyset, \lambda \theta \ . \ v) \\
\text{eval-expr} \ (t \ \text{mem volatile} \ a) &= ((1), \lambda \theta \ . \ \text{the}(\emptyset \ t)) \\
\text{eval-expr} \ (t \ \text{tmp} \ (D, f)) &= (D, f) \\
\text{eval-expr} \ (t \ \text{unop} \ f \ e) &= \text{let} (D, f_e) = \text{eval-expr} \ t \ e \ \text{in} \ (D, \lambda \theta \ . \ f \ (f_e \ \theta)) \\
\text{eval-expr} \ (t \ \text{binop} \ f \ e_1 \ e_2) &= \text{let} (D_1, f_1) = \text{eval-expr} \ t \ e_1; \\
&\quad (D_2, f_2) = \text{eval-expr} \ (t + used-tmps \ e_1) \ e_2 \\
&\quad \text{in} \ (D_1 \cup D_2, \lambda \theta \ . \ f \ (f_1 \ \theta) (f_2 \ \theta))
\end{align*}
\]

Fig. 17: Expression evaluation

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 expression $e$ followed by the corresponding store operation. Note that the ownership annotations can depend on the temporaries and thus can take the calculated address into account.

Execution of compare and swap **CAS** involves evaluation of three expressions, the address $a$ the compare value $c_e$ and the swap value $s_e$. It is finally mapped to the read-modify-write instruction **RMW** of the memory system. Recall that execution of **RMW** first stores the memory content at address $a$ to the specified temporary. The condition compares this value with the result of evaluating $c_e$ and writes swap value $s_e$ 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 ≠ Tmp sop  a' = Tmp (eval-expr t a)  t' = t + used-tmps a  \( is = \text{issue-expr t a} \)

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

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

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

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

\( \vartheta \vdash (\text{CAS } a \ c, a' \ s, A \ L \ R \ W, t) \rightarrow_p ((\text{CAS } a' \ c, a \ s, A \ L \ R \ W, t'), \ is) \)

\( D_c \subseteq \text{dom } \vartheta \)  \( \text{eval-expr } t \ s, a = (D, t) \)  \( t' = t + \text{used-tmps } s, \ cond = (\lambda \vartheta. \text{the } (\vartheta \ t') = c \ \vartheta) \)

\( \\text{ret} = (\lambda v_1, v_2, v_1) \)  \( is = \text{issue-expr } t \ s, a \ \oplus \ [(\text{RMW } (a \ \vartheta) t' (D, f) \ \text{cond } \text{ret} (A \ \vartheta) (L \ \vartheta) (R \ \vartheta) (W \ \vartheta)] \)

\( \vartheta \vdash (\text{CAS } (\text{TMP} (D_c, a)) (\text{TMP} (D_c, c)) s, t \ s, A \ L \ R \ W, t) \rightarrow_p ((\text{Skip}, t \ s), \ is) \)

\( \vartheta \vdash (\text{SEQ } s_1 s_2, t) \rightarrow_p ((\text{SEQ } s_1' s_2', t'), \ is) \)

\( \vartheta \vdash (\text{SEQ SKIP } s_2, t) \rightarrow_p ((s_2, t), []) \)

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

\( \vartheta \vdash (\text{COND } e \ s, t) \rightarrow_p ((\text{COND } e' s, t'), \ is) \)

\( D \subseteq \text{dom } \vartheta \)  \( \text{isTrue } (e \ \vartheta) \)

\( \vartheta \vdash (\text{COND } (\text{TMP } (D, e)) s, t) \rightarrow_p ((s_1, t), []) \)

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

\( \vartheta \vdash (\text{COND } (\text{TMP } (D, e)) s, t) \rightarrow_p ((s_2, t), []) \)

\( \vartheta \vdash (\text{While } e \ s, t) \rightarrow_p ((\text{COND } e \ (\text{SEQ } s \ (\text{While } e \ s)) \text{Skip}, t), []) \)

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

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

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

To instantiate Theorem 2 with PIMP we define the invariant parameter valid, which has to be maintained by all transitions of PIMP, the memory system and the store buffer. Let \( \vartheta \) be the valuation of temporaries in the current configuration, for every thread configuration \( \text{ts}_b[i] = ((s, t), i, \vartheta, \text{sb}, \mathcal{D}, \mathcal{O}) \) where \( i < |\text{ts}_b| \) we require: (i) The domain of all intermediate \( \text{Tmp}(D, f) \) expressions in statement \( s \) is below counter \( t \). (ii) All temporaries in the memory system including the store buffer are below counter \( t \). (iii) All temporaries less than counter \( t \) are either already defined in the temporaries \( \vartheta \) 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 oprations 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 additonal 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 mutal-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 ordiinary volatile writes without any flushing)
and other processors read, and each processor waits for all processors to set their flags
before continuing past the barrier. This is not sequentially consistent – each processor
might see his own flag set and later see all other flags clear – but it is still correct.

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

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

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

Acknowledgements

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

A Appendix

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

theory ReduceStoreBuffer
imports Main
begin

A.1 Memory Instructions

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

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

locale valid-sop =
fixes sop :: sop
assumes valid-sop: \( \forall D f \emptyset. \)
\[
[sop=(D,f); D \subseteq \text{dom } \emptyset] \quad \quad \Rightarrow \quad \quad f \emptyset = f (\emptyset|'D)
\]

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

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)
(proof)

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)
(proof)

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)
(proof)

lemma list-update-id′: v = xs ! i ⇒ xs[ i := v ] = xs
(proof)

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 :: tmpls ⇒ 'p ⇒ 'p × instrs ⇒ bool (- ⊢ - →p - [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 × tmpls × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒  
(instrs × tmpls × 'sb × memory × 'dirty × 'owns × 'rels × 'shared) ⇒ bool
and  
program-step :: tmpls ⇒ 'p ⇒ 'p × instrs ⇒ bool  
fixes  
record :: 'p ⇒ 'p ⇒ instrs ⇒ 'sb ⇒ 'sb

begin

inductive concurrent-step ::  
   ('p,'sb,'dirty,'owns,'rels,'shared) global-config ⇒ ('p,'sb,'dirty,'owns,'rels,'shared)
global-config ⇒ bool  
(- ⇒ - [60,60] 100)

where  
Program:  
[i < length ts; tsl!i = (p,is,∅,sb,D,O,R);  
   ∅+p →p (p′,is')] ]⇒
   (ts,m,S) ⇒ (ts[i]=(p′,is′,∅,record p p′ is′ sb,D,O,R)],m,S)

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

| StoreBuffer:  
[i < length ts; tsl!i = (p,is,∅,sb,D,O,R);  
   (m,sb,O,R,S) →sb (m′,sb′,O′,R′,S′)]  
⇒
   (ts,m,S) ⇒ (ts[i]=(p,is,∅,sb′,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** (m,sb,O,R,S) (m',sb',O',R',S')
shows \( \forall ts. i < \text{length} \ ts \Rightarrow ts[i] = (p,is,\emptyset,\text{sb},D,O,R) \Rightarrow \)
concurrent-step** (ts,m,S) (ts[i:=(p,is,\emptyset,\text{sb}',D,O',R')],m',S')
⟨proof⟩

lemma step-preserves-length-ts:
assumes step: (ts,m,S) \( \Rightarrow \) (ts',m',S')
shows length ts' = length ts
⟨proof⟩
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
\( S \oplus_{W} S \equiv (\lambda a. \text{if} \ a \in S \text{ then } \text{Some} (a \in W) \text{ else } S a) \)

definition restrict-shared:: shared \( \Rightarrow \) addr set \( \Rightarrow \) addr set \( \Rightarrow \) shared (- \( \ominus_{-} \) - [51,1000,50] 51)
where
\( S \ominus_{A} L \equiv (\lambda a. \text{if} \ a \in L \text{ then } \text{None} \text{ else } (\text{case } S \ a \text{ of } \text{None } \Rightarrow \text{None} \text{ | Some writeable } \Rightarrow \text{Some} (a \in A \lor \text{writeable}))) \)

definition read-only :: shared \( \Rightarrow \) addr set
where
read-only S \equiv \{ a. (S a = \text{Some False}) \}

definition shared-le:: shared \( \Rightarrow \) shared \( \Rightarrow \) bool (infix \( \subseteq_{s} \) 50)
where
\( m_{1} \subseteq_{s} m_{2} \equiv m_{1} \subseteq_{m} m_{2} \land \text{read-only} m_{1} \subseteq \text{read-only} m_{2} \)

lemma shared-leD: m_{1} \subseteq_{s} m_{2} \Rightarrow m_{1} \subseteq_{m} m_{2} \land \text{read-only} m_{1} \subseteq \text{read-only} m_{2}
⟨proof⟩

lemma shared-le-map-le: m_{1} \subseteq_{s} m_{2} \Rightarrow m_{1} \subseteq_{m} m_{2}
⟨proof⟩

lemma shared-le-read-only-le: m_{1} \subseteq_{s} m_{2} \Rightarrow \text{read-only} m_{1} \subseteq \text{read-only} m_{2}
⟨proof⟩

lemma dom-augment [simp]: dom (m \( \oplus_{W} \) S) = dom m \cup S
⟨proof⟩

lemma augment-empty [simp]: S \( \oplus_{x} \) {} = S
⟨proof⟩
lemma inter-neg [simp]: $X \cap -L = X - L$

(\textit{proof})

lemma dom-restrict-shared [simp]: $\text{dom } (m \ominus_A L) = \text{dom } m - L$

(\textit{proof})

lemma restrict-shared-UNIV [simp]: $(m \ominus_A \text{UNIV}) = \text{Map.empty}$

(\textit{proof})

lemma restrict-shared-empty [simp]: $(\text{Map.empty} \ominus_A L) = \text{Map.empty}$

(\textit{proof})

lemma restrict-shared-in [simp]: $a \in L \implies (m \ominus_A L) a = \text{None}$

(\textit{proof})

lemma restrict-shared-out: $a \not\in L \implies (m \ominus_A L) a = \text{map-option \{\lambda\text{writeable}. \{a \in A \or \text{writeable}\}\}}(m a)$

(\textit{proof})

lemma restrict-shared-out\textsuperscript{'} [simp]: $a \not\in L \implies m a = \text{Some \text{writeable}} \implies (m \ominus_A L) a = \text{Some \{a \in A \or \text{writeable}\}}$

(\textit{proof})

lemma augment-mono-map\textsuperscript{'}: $A \subseteq_m B \implies (A \oplus_x C) \subseteq_m (B \oplus_x C)$

(\textit{proof})

lemma augment mono map: $A \subseteq_s B \implies (A \oplus_x C) \subseteq_s (B \oplus_x C)$

(\textit{proof})

lemma restrict mono map: $A \subseteq_s B \implies (A \ominus_x C) \subseteq_s (B \ominus_x C)$

(\textit{proof})

lemma augment mono aux: $\text{dom } A \subseteq \text{dom } B \implies \text{dom } (A \ominus_x C) \subseteq \text{dom } (B \ominus_x C)$

(\textit{proof})

lemma restrict mono aux: $\text{dom } A \subseteq \text{dom } B \implies \text{dom } (A \ominus x C) \subseteq \text{dom } (B \ominus x C)$

(\textit{proof})

lemma read-only mono: $S \subseteq_m S' \implies a \in \text{read-only } S \implies a \in \text{read-only } S'$

(\textit{proof})

lemma in-read-only restrict conv: $a \in \text{read-only } (S \ominus_A L) = (a \in \text{read-only } S \land a \not\in L \land a \not\in A)$

(\textit{proof})

35
lemma in-read-only-augment-conv: \( a \in \text{read-only} (S \oplus W R) = (\text{if } a \in R \text{ then } a \notin W \text{ else } a \in \text{read-only } S) \)

\( \langle \text{proof} \rangle \)

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

lemma read-only-dom: \( \text{read-only } S \subseteq \text{dom } S \)

\( \langle \text{proof} \rangle \)

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

\( \langle \text{proof} \rangle \)

definition augment-rels:: addr set \( \Rightarrow \) addr set \( \Rightarrow \) rels \( \Rightarrow \) rels

where

augment-rels S R \( \mathcal{R} \) = (\( \lambda a \). if \( a \in R \)

then (case \( \mathcal{R} \) a of

None \( \Rightarrow \) Some (\( a \in S \))

| Some s \( \Rightarrow \) Some (s \( \land \) (\( a \in S \))))

else \( \mathcal{R} \) a)

declare domIff [iff del]

A.3 Memory Transitions

locale gen-direct-memop-step =

fixes emp::'rels and aug::owns \( \Rightarrow \) rel \( \Rightarrow \) 'rels \( \Rightarrow \) 'rels

begin

inductive gen-direct-memop-step :: (instrs \times \text{tmps} \times \text{unit} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{'rels} \times \text{shared} ) \Rightarrow

(instrs \times \text{tmps} \times \text{unit} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{'rels} \times \text{shared} ) \Rightarrow \text{bool}

(- \rightarrow - [60,60] 100)

where

Read: (Read volatile a t \( \# \) is,\( \varnothing \), x, m,\( D \), \( O \), \( \mathcal{R} \), \( S \)) \( \rightarrow \)

(is, \( \varnothing \) (t\( \rightarrow \)m a), x, m,\( D \), \( O \), \( \mathcal{R} \), \( S \))

| WriteNonVolatile:

(Write False a (D,f) A L R W\( \# \)is, \( \varnothing \), x, m,\( D \), \( O \), \( \mathcal{R} \), \( S \)) \( \rightarrow \)

(is, \( \varnothing \), x, m(a:= f \( \varnothing \)), \( D \), \( O \), \( \mathcal{R} \), \( S \))

| WriteVolatile:

(Write True a (D,f) A L R W\( \# \) is, \( \varnothing \), x, m,\( D \), \( O \), \( \mathcal{R} \), \( S \)) \( \rightarrow \)

(is, \( \varnothing \), x, m(a:=f \( \varnothing \)), True, \( O \cup A - R \), emp, \( S \oplus W R \ominus A L \))

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

| RMWReadOnly: |
| \[ \neg \text{cond } (\phi(t \rightarrow m \ a)) \] \implies (RMW a t (D,f) cond ret A L R W # is, \( \phi \), \( x \), \( m \), \( D \), \( O \), \( R \), \( S \)) \rightarrow (is, \( \phi(t \rightarrow m \ a) \), x,m, False, \( O \), emp, \( S \)) |

| RMWWrite: |
| \[ \text{cond } (\phi(t \rightarrow m \ a)) \] \implies (RMW a t (D,f) cond ret A L R W# is, \( \phi \), \( x \), \( m \), \( D \), \( O \), \( R \), \( S \)) \rightarrow (is, \( \phi(t \rightarrow ret \ (m \ a)) \), x, m(a\leftarrow f(\phi(t \rightarrow m \ a)) \)), False, \( O \cup A - R \), emp, \( S \oplus W \, R \ominus A \, L \) |

| Ghost: |
| (Ghost A L R W # is, \( \phi \), \( x \), \( m \), \( D \), \( O \), \( R \), \( S \)) \rightarrow (is, \( \phi \), \( x \), \( m \), \( D \), \( O \cup A - R \), aug (dom \( S \)) \; R \; R \; \), \( S \oplus W \, R \ominus A \, L \) |

end

interpretation direct-memop-step: gen-direct-memop-step Map.empty augment-rels ⟨proof⟩

term direct-memop-step.gen-direct-memop-step
abbreviation direct-memop-step :: (instrs × tmmps × unit × memory × bool × owns × rels × shared ) \Rightarrow (instrs × tmmps × unit × memory × bool × owns × rels × shared ) \Rightarrow bool
\((- \rightarrow - [60,60] \, 100)\)

where
direct-memop-step \equiv \text{direct-memop-step.gen-direct-memop-step}

term x \rightarrow Y

abbreviation direct-memop-steps ::
\((\text{instrs} \times \text{tmmps} \times \text{unit} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{rels} \times \text{shared}) \Rightarrow \text{bool}
\((- \rightarrow^* - [60,60] \, 100)\)

where
direct-memop-steps \equiv (\text{direct-memop-step})^*^*

term x \rightarrow^* Y

interpretation virtual-memop-step: gen-direct-memop-step () (\(\lambda S \, \; R \, \; . \; ()\)) ⟨proof⟩

abbreviation virtual-memop-step :: (instrs × tmmps × unit × memory × bool × owns × unit × shared ) \Rightarrow (\(\text{instrs} \times \text{tmmps} \times \text{unit} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{unit} \times \text{shared}\) \Rightarrow bool
\((- \rightarrow_\nu - [60,60] \, 100)\)

where
virtual-memop-step \equiv \text{virtual-memop-step.gen-direct-memop-step}
term \( x \rightarrow^* Y \)

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

\[
(\text{instrs} \times \text{tmps} \times \text{unit} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{unit} \times \text{shared}) \Rightarrow \text{bool}
\]

\(- \rightarrow^* - [60,60] 100\)

**where**

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

\[
\text{term} \ x \rightarrow^* Y
\]

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

**assumes** step:

\[\text{(is, } \theta, \ x, \ m, \ D, \ O, \ R, \ S) \rightarrow (\text{is}', \ \theta', \ x', \ m', \ D', \ O', \ R', \ S')\]

**shows** (is, \( \theta, x, m, D, O, () S \)) \( \rightarrow^* (\text{is}', \ \theta', x', m', D', O', (), S')\)

\[
\langle \text{proof} \rangle
\]

A.4 Safe Configurations of Virtual Machines

**inductive** safe-direct-memop-state :: owns list \( \Rightarrow \text{nat} \Rightarrow (\text{instrs} \times \text{tmps} \times \text{memory} \times \text{bool} \times \text{owns} \times \text{shared}) \Rightarrow \text{bool}
\]

\(- \vdash - \sqrt{[60,60,60]} 100\)

**where**

Read: \([a \in O \lor a \in \text{read-only} S \lor (\text{volatile} \land a \in \text{dom} S); \]

volatile \( \rightarrow \neg D \]

\(\vdash O_s,i \vdash (\text{Read volatile} a \ t \ # \ \text{is}, \ \theta, m, D, O, S)\)

| WriteNonVolatile:
| \([a \in O; a \notin \text{dom} S]\]
| \(\vdash O_s,i \vdash (\text{Write False} a (D,f) A L R W # \text{is}, \ \theta, m, D, O, S)\)

| WriteVolatile:
| \[\forall j < \text{length } O_s. i \neq j \longrightarrow a \notin O_s!j; \]
| \(A \subseteq \text{dom } S \cup O; \ L \subseteq A; \ R \subseteq O; A \cap R = \{\}; \]
| \(\forall j < \text{length } O_s. i \neq j \longrightarrow A \cap O_s!j = \{\}; \]
| \(a \notin \text{read-only } S \]
| \(\vdash O_s,i \vdash (\text{Write True} a (D,f) A L R W # \text{is}, \ \theta, m, D, O, S)\)

| Fence:
| \(O_s,i \vdash (\text{Fence} # \text{is}, \ \theta, m, D, O, S)\)

| Ghost:
| \([A \subseteq \text{dom } S \cup O; L \subseteq A; R \subseteq O; A \cap R = \{\}; \]
| \(\forall j < \text{length } O_s. i \neq j \longrightarrow A \cap O_s!j = \{\}]\)
\[

\sigma, i \vdash (\text{Ghost } A \ L \ R \ W\# \text{ is, } \varnothing, m, D, O, S) \vee
\]

| RMWReadOnly: |
|-cond (\varnothing(t \rightarrow m a)); a \in O \vee a \in \text{dom } S]-\]\n\[
\sigma, i \vdash (\text{RMW a t (D,f) cond ret A L R W\# is, } \varnothing, m, D, O, S) \vee
\]

| RMWWrite: |
| \text{cond (\varnothing(t \rightarrow m a))}; \forall j < \text{length } O s. i \neq j \rightarrow a \notin O s[j]; \forall j < \text{length } O s. i \neq j \rightarrow \text{A } \cap \text{O}s[j] = \{\}; a \notin \text{read-only } S]-\]\n\[
\sigma, i \vdash (\text{RMW a t (D,f) cond ret A L R W\# is, } \varnothing, m, D, O, S) \vee
\]

| Nil: |
| \sigma, i \vdash ([], \varnothing, m, D, O, S) \vee
\]

\textit{inductive} \text{ safe-delayed-direct-memop-state :: owns list } \Rightarrow \text{rels list } \Rightarrow \text{nat } \Rightarrow
\]
\text{(instrs } \times \text{tmpts } \times \text{memory } \times \text{owns } \times \text{shared}) \Rightarrow \text{bool}
\]
\text{(-, -, -) - } \Rightarrow [60, 60, 60, 60] 100

\textbf{where}

\text{Read: } [a \in O \lor a \in \text{read-only } S \lor (\text{volatile } \land a \in \text{dom } S)\];
\]
\forall j < \text{length } O s. i \neq j \rightarrow (\text{R}s[i]) a \neq \text{Some False; } - \text{ no release of unshared address}
\]
\text{volatile } \rightarrow (\forall j < \text{length } O s. i \neq j \rightarrow a \notin (\text{R}s[j]));
\]
\text{volatile } \rightarrow - D ]
\]
\[
\Rightarrow \sigma, \text{R}s, i \vdash (\text{Read volatile a t } \# \text{ is, } \varnothing, m, D, O, S) \vee
\]

| WriteNonVolatile: |
| [a \in O; a \notin \text{dom } S; \forall j < \text{length } O s. i \neq j \rightarrow a \notin (\text{R}s[j])] ]
\]
\[
\Rightarrow \sigma, \text{R}s, i \vdash (\text{Write False a (D,f) A L R W\#is, } \varnothing, m, D, O, S) \vee
\]

| WriteVolatile: |
| [\forall j < \text{length } O s. i \neq j \rightarrow a \notin (O s[j] \cup \text{dom } (R s[j])); \forall j < \text{length } O s. i \neq j \rightarrow \text{A } \cap \text{O}s[j] \cup \text{dom } (R s[j]) = \{\}; a \notin \text{read-only } S] ]
\]
\[
\Rightarrow \sigma, \text{R}s, i \vdash (\text{Write True a (D,f) A L R W\# is, } \varnothing, m, D, O, S) \vee
\]

| Fence: |
| \sigma, \text{R}s, i \vdash (\text{Fence } \# \text{ is, } \varnothing, m, D, O, S) \vee
\]

| Ghost: |
| [\text{A } \subseteq \text{dom } S \cup O; L \subseteq A; R \subseteq O; A \cap R = \{\}; \forall j < \text{length } O s. i \neq j \rightarrow \text{A } \cap \text{O}s[j] \cup \text{dom } (R s[j]) = \{\}] ]
\]
\[
\Rightarrow \sigma, \text{R}s, i \vdash (\text{Ghost } A \ L \ R \ W\# \text{ is, } \varnothing, m, D, O, S) \vee
\]

39
RMWReadOnly:
\[ \neg \text{cond (}\theta(t\mapsto m a))\); a \in O \lor a \in \text{dom } S; \\
\forall j < \text{length } Os. i \neq j \rightarrow (Rs\!j) a \neq \text{Some False} — \text{no release of unshared address}] \\
\implies \\
Os, Rs, i \vdash (\text{RMW} a t (D,f) \text{ cond ret } A L R W# \text{ is } \theta, m, D, O, S) \sqrt{}

RMWWrite:
\[ \text{cond (}\theta(t\mapsto m a))\); a \in O \lor a \in \text{dom } S; \\
\forall j < \text{length } Os. i \neq j \rightarrow a \notin (Os\!j \cup \text{dom } (Rs\!j)); \\
A \subseteq \text{dom } S \cup O; L \subseteq A; R \subseteq O; A \cap R = \{\}; \\
\forall j < \text{length } Os. i \neq j \rightarrow A \cap (Os\!j \cup \text{dom } (Rs\!j)) = \{\}; \\
a \notin \text{read-only } S] \\
\implies \\
Os, Rs, i \vdash (\text{RMW} a t (D,f) \text{ cond ret } A L R W# \text{ is } \theta, m, D, O, S) \sqrt{}

Nil: \\
Os, Rs, i \vdash ([], \theta, m, D, O, S) \sqrt{}

lemma memop-safe-delayed-implies-safe-free-floating:
assumes safe-delayed: Os, Rs, i \vdash (is, \theta, m, D, O, S) \sqrt{}
shows Os, i \vdash (is, \theta, m, D, O, S) \sqrt{}
(proof)

lemma memop-empty-rels-safe-free-floating-implies-safe-delayed:
assumes safe: Os, i \vdash (is, \theta, m, D, O, S) \sqrt{}
assumes empty: \forall R \in \text{set } Rs. R = \text{Map.empty}
assumes leq: length Os = length Rs
assumes unowned-shared: (\forall a. (\forall i < \text{length } Os. a \notin (Os\!i)) \rightarrow a \in \text{dom } S)
assumes Os-i: Os\!i = O
shows Os, Rs, i \vdash (is, \theta, m, D, O, S) \sqrt{}
(proof)

inductive id-storebuffer-step:
(memory \times \text{unit} \times \text{owns} \times \text{rels} \times \text{shared}) \Rightarrow (memory \times \text{unit} \times \text{owns} \times \text{rels} \times \text{shared}) \\
\Rightarrow \text{bool} (\rightarrow \rightarrow [60,60] 100)
where
Id: (m,x,O,R,S) \rightarrow_1 (m,x,O,R,S)

definition empty-storebuffer-step:: (memory \times \text{'sb} \times \text{'owns} \times \text{'rels} \times \text{'shared}) \Rightarrow (memory \times \text{'sb} \times \text{'owns} \times \text{'rels} \times \text{'shared}) \Rightarrow \text{bool}
where
empty-storebuffer-step c c' = False

context program
begin

abbreviation direct-concurrent-step ::
(‘p,unit,bool,owns,rels,shared) global-config \Rightarrow (‘p,unit,bool,owns,rels,shared)
global-config \Rightarrow \text{bool}
\[
\text{where}
\]
direct-concurrent-step \equiv
\text{computation.concurrent-step} \quad \text{direct-memop-step} \quad \text{gen-direct-memop-step}
empty-storebuffer-step \quad \text{program-step}

(\lambda p \ p' \text{ is sb. sb})

\text{abbreviation} \quad \text{direct-concurrent-steps}::
(\text{'} p, \text{unit}, \text{bool}, \text{owns}, \text{rels}, \text{shared}) \quad \text{global-config} \Rightarrow (\text{'} p, \text{unit}, \text{bool}, \text{owns}, \text{rels}, \text{shared})
global-config \Rightarrow \text{bool}
(- \Rightarrow d^* - [60,60] 100)

\text{where}
direct-concurrent-steps == direct-concurrent-step^{**}

\text{abbreviation} \quad \text{virtual-concurrent-step} ::
(\text{'} p, \text{unit}, \text{bool}, \text{owns}, \text{unit}, \text{shared}) \quad \text{global-config} \Rightarrow (\text{'} p, \text{unit}, \text{bool}, \text{owns}, \text{unit}, \text{shared})
global-config \Rightarrow \text{bool}
(- \Rightarrow v^* - [100,60] 100)

\text{where}
virtual-concurrent-step \equiv
\text{computation.concurrent-step} \quad \text{virtual-memop-step} \quad \text{gen-direct-memop-step}
empty-storebuffer-step \quad \text{program-step}

(\lambda p \ p' \text{ is sb. sb})

\text{abbreviation} \quad \text{virtual-concurrent-steps}::
(\text{'} p, \text{unit}, \text{bool}, \text{owns}, \text{unit}, \text{shared}) \quad \text{global-config} \Rightarrow (\text{'} p, \text{unit}, \text{bool}, \text{owns}, \text{unit}, \text{shared})
global-config \Rightarrow \text{bool}
(- \Rightarrow v^* - [60,60] 100)

\text{where}
virtual-concurrent-steps == virtual-concurrent-step^{**}

\text{term} \quad x \Rightarrow v \quad Y
\text{term} \quad x \Rightarrow d \quad Y

\text{term} \quad x \Rightarrow d^* \quad Y
\text{term} \quad x \Rightarrow v^* \quad Y

\text{end}

\text{definition}
safe-reach step safe cfg \equiv
\forall \text{cfg', step}^{**} \text{cfg cfg'} \longrightarrow \text{safe cfg'}

\text{lemma} \quad \text{safe-reach-safe-refl:} \quad \text{safe-reach step safe cfg} \Longrightarrow \text{safe cfg}
\langle \text{proof} \rangle

\text{lemma} \quad \text{safe-reach-safe-rtrancl:} \quad \text{safe-reach step safe cfg} \Longrightarrow \text{step}^{**} \text{cfg cfg'} \Longrightarrow \text{safe cfg'}
\langle \text{proof} \rangle
lemma safe-reach-steps: safe-reach step safe cfg ⇒ step² cfg cfg' ⇒ safe-reach step safe cfg'
(proof)

lemma safe-reach-step: safe-reach step safe cfg ⇒ step cfg cfg' ⇒ safe-reach step safe cfg'
(proof)

context program
begin

abbreviation safe-reach-direct ≡ safe-reach direct-concurrent-step

lemma safe-reac-direct-def':
safe-reach-direct safe cfg ≡
∀ cfg', cfg ⇒ₜ² cfg' → safe cfg'
(proof)

abbreviation safe-reach-virtual ≡ safe-reach virtual-concurrent-step

lemma safe-reac-virtual-def':
safe-reach-virtual safe cfg ≡
∀ cfg'. cfg ⇒ₜₙ cfg' → safe cfg'
(proof)
end

definition safe-free-flowing cfg ≡ let (ts,m,S) = cfg
  in (∀ i < length ts. let (p,is,θ,x,D,O,R) = ts!i in
    map owned ts,i ⊢ (is,θ,m,D,O,S)✓)

lemma safeE: [safe-free-flowing (ts,m,S);i<length ts; ts!i=(p,is,θ,x,D,O,R)]
  ⇒ map owned ts,i ⊢ (is,θ,m,D,O,S)✓
(proof)

definition safe-delayed cfg ≡ let (ts,m,S) = cfg
  in (∀ i < length ts. let (p,is,θ,x,D,O,R) = ts!i in
    map owned ts,map released ts,i ⊢ (is,θ,m,D,O,S)✓)

lemma safe-delayedE: [safe-delayed (ts,m,S);i<length ts; ts!i=(p,is,θ,x,D,O,R)]
  ⇒ map owned ts,map released ts,i ⊢ (is,θ,m,D,O,S)✓
(proof)

definition remove-rels ≡ map (λ(p,is,θ,sb,D,O,R). (p,is,θ,sb,D,O,()())

theorem (in program) virtual-simulates-direct-step:
assumes step: \((ts,m,S) \Rightarrow_d (ts',m',S')\)

shows \((remove-rels ts,m,S) \Rightarrow_{\nu} (remove-rels ts',m',S')\)

(proof)

lemmas converse-rtranclp-induct-sbh-steps = converse-rtranclp-induct
[of - \((ts,m,S) \Rightarrow_d (ts',m',S')\), 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 \((remove-rels ts,m,S) \Rightarrow_{\nu}^* (remove-rels ts',m',S')\)

(proof)

locale simple-ownership-distinct =

fixes ts::\(\langle \theta^p, sb', \theta',\text{owns}, \text{rels} \rangle\) thread-config list

assumes simple-ownership-distinct:
\[\forall i \ j \ p \ is \ \theta \ \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \mathcal{S} \ \mathcal{sb}.
\begin{align*}
& [i < \text{length } ts; \ j < \text{length } ts; \ i \neq j; \\
& tsli = (p, i, j, \theta, \mathcal{O}, \mathcal{R}, \mathcal{D}) ; \ tslj = (p, j, i, \theta, \mathcal{O}, \mathcal{R}, \mathcal{D})] \\
\Rightarrow & \mathcal{O}_i \cap \mathcal{O}_j = \{\}
\end{align*}\]

lemma (in simple-ownership-distinct)
simple-ownership-distinct-nth-update:
\[\forall i \ j \ is \ \theta \ \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \mathcal{S} \ \mathcal{sb}.
\begin{align*}
& [i < \text{length } ts; \ tsli = (p, i, j, \theta, \mathcal{O}, \mathcal{R}, \mathcal{D}) ] \\
\forall & j < \text{length } ts; \ i \neq j \longrightarrow (\text{let } ts' = tsli \in (\mathcal{O}' \cap (\mathcal{O}_j) = \{\})] \\
\Rightarrow & \text{simple-ownership-distinct } (ts[i := (p', \text{is}', \theta', \mathcal{D}', \theta', \mathcal{O}', \mathcal{R}')])
\end{align*}\]

(proof)

locale read-only-unowned =

fixes \(\mathcal{S}::\text{shared and ts::\(\langle \theta^p, sb', \theta',\text{owns}, \text{rels} \rangle\) thread-config list}

assumes read-only-unowned:
\[\forall i \ p \ is \ \theta \ \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \mathcal{S} \ \mathcal{sb}.
\begin{align*}
& [i < \text{length } ts; \ tsli = (p, i, j, \theta, \mathcal{O}, \mathcal{R}, \mathcal{D})] \\
\Rightarrow & \mathcal{O} \cap \text{read-only } \mathcal{S} = \{\}
\end{align*}\]

lemma (in read-only-unowned)
read-only-unowned-nth-update:
\[\forall i \ p \ is \ \theta \ \mathcal{O} \ \mathcal{R} \ \mathcal{D} \ \mathcal{acq} \ \theta \ \mathcal{sb}.
\begin{align*}
& [i < \text{length } ts; \ \mathcal{O} \cap \text{read-only } \mathcal{S} = \{\}] \\
\Rightarrow & \text{read-only-unowned } \mathcal{S} (ts[i := (p, i, j, \theta, \mathcal{D}, \theta, \mathcal{O}, \mathcal{R})])
\end{align*}\]

(proof)

locale unowned-shared =

fixes \(\mathcal{S}::\text{shared and ts::\(\langle \theta^p, sb', \theta',\text{owns}, \text{rels} \rangle\) thread-config list}

assumes unowned-shared: \(- \cup ((\lambda(\theta, \theta, \theta, \theta, \mathcal{O}, \mathcal{S}) \setminus \text{set } ts) \subseteq \text{dom } \mathcal{S})\)
lemma (in unowned-shared)
unowned-shared-nth-update:
assumes i-bound: i < length ts
assumes ith: ts!i = (p, is, xs, sb, D, O, R)
assumes subset: O ⊆ O'
s shows unowned-shared S (ts[i := (p', is', xs', sb', D', O', R')])
proof

lemma (in unowned-shared) a-unowned-by-others-owned-or-shared:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, xs, sb, D, O, R)
assumes a-unowned-others:
∀ j < length (map owned ts). i ≠ j —→
(let O_j = (map owned ts)!j in a /∈ O_j)
s shows a ∈ O ∨ a ∈ dom S
proof

lemma (in unowned-shared) unowned-shared':
assumes notin: ∀ i < length ts. a /∈ owned (ts!i)
s shows a ∈ dom S
proof

lemma unowned-shared-def': unowned-shared S ts = (∀ a. (∀ i < length ts, a /∈ owned (ts!i)) —→ a ∈ dom S)
proof

definition
initial cfg ≡ let (ts, m, S) = cfg
 in unowned-shared S ts ∧
 ( ∀ i < length ts. let (p, is, θ, xs, D, O, R) = ts!i in
 R = Map.empty )

lemma initial-empty-rels: initial (ts, m, S) —→ ∀ R ∈ set (map released ts). R = Map.empty
proof

lemma initial-unowned-shared: initial (ts, m, S) —→ unowned-shared S ts
proof

lemma initial-safe-free-flowing-implies-safe-delayed:
assumes init: initial c
assumes safe: safe-free-flowing c
shows safe-delayed c
proof

locale program-progress = program +
assumes progress: θ ⊢ p → p (p', is') —→ p' ≠ p ∨ is' ≠ []
The assumption ‘progress’ could be avoided if we introduce stuttering steps in lemma undo-local-step or make the scheduling of threads explicit, such that we can directly express that ‘thread i does not make a step’. lemma (in program-progress) undo-local-step:

assumes step: (ts,m,S) \rightarrow_d (ts',m',S')
assumes i-bound: i < length ts
assumes unchanged: ts!i = ts'!i
assumes safe-delayed-undo: safe-delayed (u-ts,u-m,u-shared) — proof should also work with weaker safe-free-flowing
assumes leq: length u-ts = length ts
assumes others-same: \forall j < length ts. j\neq i \rightarrow u-ts!j = ts!j
assumes u-ts-i: u-ts!i=(u-p,u-is,u-tmps,u-x,u-dirty,u-owns,u-rels)
assumes u-m-other: \forall a. a \not\in 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 \not\in u-owns \rightarrow a \not\in owned (ts!i) \rightarrow u-shared a = S a
assumes dist: simple-ownership-distinct u-ts
assumes dist-ts: simple-ownership-distinct ts

shows \exists u-ts' u-shared' u-m'. (u-ts,u-m,u-shared) \Rightarrow_d (u-ts',u-m',u-shared') \land

— thread i is unchanged
u-ts'!i = u-ts!i \land
(\forall a \in u-owns. u-shared' a = u-shared a) \land
(\forall a \in u-owns. S' a = S a) \land
(\forall a \in u-owns. u-m' a = u-m a) \land
(\forall a \in u-owns. m' a = m a) \land

— other threads are simulated
(\forall j < length ts. j\neq i \rightarrow u-ts'!j = ts!j) \land
(\forall a. a \not\in u-owns \rightarrow a \not\in owned (ts!i) \rightarrow u-shared' a = S' a) \land
(\forall a. a \not\in u-owns \rightarrow u-m' a = m' a)

⟨proof⟩

theorem (in program) safe-step-preserves-simple-ownership-distinct:

assumes step: (ts,m,S) \rightarrow_d (ts',m',S')
assumes safe: safe-delayed (ts,m,S)
assumes dist: simple-ownership-distinct ts

shows simple-ownership-distinct ts'

⟨proof⟩

theorem (in program) safe-step-preserves-read-only-unowned:

assumes step: (ts,m,S) \rightarrow_d (ts',m',S')
assumes safe: safe-delayed (ts,m,S)
assumes ro-unowned: read-only-unowned S ts

shows read-only-unowned S' ts'

⟨proof⟩

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)

⟨proof⟩
assumes dist: simple-ownership-distinct ts
assumes unowned-shared: unowned-shared $S$ ts
shows unowned-shared $S'$ ts'
(proof)

locale program-trace = program +
fixes c — enumeration of configurations: c n $\Rightarrow_d$ c (n + 1) ... $\Rightarrow_d$ c (n + k)
fixes n::nat — starting index
fixes k::nat — steps

assumes step: $\forall l. 1 < k \implies c (n+l) \Rightarrow_d c (n + (Suc l))$

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

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

lemma split-less-Suc: ($\forall x<\text{Suc} k. P x$) = (P k $\land$ ($\forall x<k. P x$))
(proof)

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

lemma (in program) steps-to-trace:
assumes steps: x $\Rightarrow_d^*$ y
shows $\exists c k. \text{trace} c 0 k \land c 0 = x \land c k = y$
(proof)

lemma (in program) trace-preserves-length-ts:
$\forall l. \text{trace} c n k \implies l \leq k \implies x \leq k \implies \text{length} (\text{fst} (c (n + l))) = \text{length} (\text{fst} (c (n + x)))$
(proof)

lemma (in program) trace-preserves-simple-ownership-distinct:
assumes dist: simple-ownership-distinct (fst (c n))
shows $\forall l. \text{trace} c n k \implies (\forall x<k. \text{safe-delayed} (c (n + x))) \implies$
$1 \leq k \implies \text{simple-ownership-distinct} (\text{fst} (c (n + l)))$
(proof)

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 $\forall l. \text{trace} c n k \implies (\forall x<k. \text{safe-delayed} (c (n + x))) \implies$
$1 \leq k \implies \text{read-only-unowned} (\text{snd} (\text{snd} (c (n + l)))) (\text{fst} (c (n + l)))$
(proof)

lemma (in program) trace-preserves-unowned-shared:
assumes dist: simple-ownership-distinct (fst (c n))
assumes ro: unowned-shared (snd (snd (c n))) (fst (c n))
shows \( \forall l. \) trace c n k \( \implies (\forall x < k. \) safe-delayed \((c (n + x)) \implies 1 \leq k \implies \) unowned-shared \((snd (snd (c (n + l)))) (fst (c (n + l)))) \)

\( \langle \text{proof} \rangle \)

\textbf{theorem (in program-progress) undo-local-steps:}  
\textbf{assumes steps: trace c n k}
\textbf{assumes c-n: c n = (ts,m,S)}
\textbf{assumes unchanged: }\forall l \leq k. \((\forall ts \ S_l \ m_l . c \ (n + l) = (ts_l,m_l,S_l) \implies ts_l\!i=ts\!i)\)
\textbf{assumes safe: safe-delayed (u-ts, u-m, u-shared)}
\textbf{assumes leq: length u-ts = length ts}
\textbf{assumes i-bound: i < length ts}
\textbf{assumes others-same: }\forall j < length ts. \ j \neq i \implies u-ts\!j = ts\!j
\textbf{assumes u-ts-i: u-ts\!i = (u-p, u-is, u-temps, u-sb, u-dirty, u-owns, u-rels)}
\textbf{assumes u-m-other: }\forall a. \ a \notin u-owns \implies u-m \ a = m \ a
\textbf{assumes u-m-shared: }\forall a. \ a \in u-owns \implies a \in dom u-shared \implies u-m \ a = m \ a
\textbf{assumes u-shared: }\forall a. \ a \notin u-owns \implies a \notin owned (ts\!i) \implies u-shared \ a = S \ a
\textbf{assumes dist: simple-ownership-distinct u-ts}
\textbf{assumes dist-ts: simple-ownership-distinct ts}
\textbf{assumes safe-orig: }\forall x. \ x < k \implies \text{safe-delayed } (c \ (n + x))
\textbf{shows }\exists c'. l. 1 \leq k \wedge \text{trace } c' n l \wedge
\( c' \ n = (u-ts, u-m, u-shared) \wedge \)
\( (\forall x \leq l. \) length \((fst (c' \ (n + x)))) = length \((fst (c \ (n + x)))) \wedge \)
\( (\forall x < l. \) safe-delayed \((c' \ (n + x))) \wedge \)
\( (1 < k \implies \neg \text{safe-delayed } (c' \ (n + l))) \wedge \)
\( (\forall x \leq l. \forall ts_x \ S_{x} \ m_{x} \ ts_{x}' \ S'_{x} \ m'_{x}. \ c \ (n + x) = (ts_{x}, m_{x}, S_{x}) \implies c' \ (n + x) = \)
\( (ts_{x}', m_{x}', S'_{x}) \implies \)
\( ts_{x}\!i=ts\!i \wedge \)
\( (\forall a \in u-owns. \ S_{x}' a = u-shared a) \wedge \)
\( (\forall a \in u-owns. \ S_{x} a = S \ a) \wedge \)
\( (\forall a \in u-owns. \ S_{x}' a = u-shared a) \wedge \)
\( (\forall a \in u-owns. \ m_{x}' a = u-m \ a) \wedge \)
\( (\forall a \in u-owns. \ m_{x} a = m \ a) \wedge \)
\( (\forall x \leq l. \forall ts_x \ S_{x} \ m_{x} \ ts_{x}' \ S'_{x} \ m'_{x}. \ c \ (n + x) = (ts_{x}, m_{x}, S_{x}) \implies c' \ (n + x) = \)
\( (ts_{x}', m_{x}', S'_{x}) \implies \)
\( (\forall j < length ts_x. \ j \neq i \implies ts_{x}\!j = ts\!j) \wedge \)
\( (\forall a. \ a \notin u-owns \implies a \notin owned (ts\!i) \implies S_{x}' a = S_{x} a) \wedge \)
\( (\forall a. \ a \notin u-owns \implies m_{x}' a = m_{x} a)) \)

\( \langle \text{proof} \rangle \)

\textbf{locale program-safe-reach-upto = program +}
\textbf{fixes n fixes safe fixes c_{0}}
\textbf{assumes safe-config: }[k \leq n; \text{trace c 0 k; c 0 = c_{0}; 1 \leq k} \implies \text{safe } (c \ l)]
abbreviation (in program)
safe-reach-upto ≡ program-safe-reach-upto program-step

lemma (in program) safe-reach-upto-le:
assumes safe: safe-reach-upto n safe c_0
assumes m-n: m ≤ n
shows safe-reach-upto m safe c_0
⟨proof⟩

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 → l ≤ k → fst (c l)!i = fst (c k)!i))
⟨proof⟩

lemma (in program) sequence-traces:
assumes trace1: trace c_1 0 k
assumes trace2: trace c_2 m l
assumes seq: c_2 m = c_1 k
assumes c-def: c = (λx. if x ≤ k then c_1 x else (c_2 (m + x − k)))
shows trace c 0 (k + l)
⟨proof⟩

theorem (in program-progress) safe-free-flowing-implies-safe-delayed:
assumes init: initial c_0
assumes dist: simple-ownership-distinct (fst c_0)
assumes read-only-unowned: read-only-unowned (snd (snd c_0)) (fst c_0)
assumes unowned-shared: unowned-shared (snd (snd c_0)) (fst c_0)
assumes safe-reach-ff: safe-reach-upto n safe-free-flowing c_0
shows safe-reach-upto n safe-delayed c_0
⟨proof⟩

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
(- →r - [60,60] 100)
where
Write_{sb}: [O'] = (if volatile then O ∪ A − R else O);
\[ S' = (\text{if volatile then } S \oplus W R \ominus A L \text{ else } S); \]
\[ \mathcal{R}' = (\text{if volatile then } \text{Map.empty else } \mathcal{R}) ] \]
\[ \implies \]
\[ (m, \text{Write}_{ab} \text{ volatile a sop v A L R W# rs, } \mathcal{O}, \mathcal{R}, S) \rightarrow_t (m(a := v), rs, \mathcal{O}', \mathcal{R}', S') \]
| Read_{sb}: (m, \text{Read}_{sb} \text{ volatile a t v#rs, } \mathcal{O}, \mathcal{R}, S) \rightarrow_t (m, rs, \mathcal{O}, \mathcal{R}, S) \]
| Prog_{sb}: (m, \text{Prog}_{sb} p p' is#rs, \mathcal{O}, \mathcal{R}, S) \rightarrow_t (m, rs, \mathcal{O}, \mathcal{R}, S) \]
| Ghost: (m, \text{Ghost}_{sb} A L R W# rs, \mathcal{O}, \mathcal{R}, S) \rightarrow_t (m, rs, \mathcal{O} \cup A - R, \text{augment-rels} (\text{dom } \mathcal{S}) \]
\[ R\ \mathcal{R}, \ S \oplus W R \ominus A L \]

**abbreviation** flush-steps::memory \times 'p store-buffer \times owns \times rels \times shared \Rightarrow \text{memory} \times 'p store-buffer \times owns \times rels \times shared \Rightarrow \text{bool}

\[ (- \rightarrow_t^* - [60,60] 100) \]

**where**

flush-steps == flush-step\(^*\)**

**term** x \(\rightarrow_t^* Y\)

**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 \times 'p store-buffer \times 'owns \times 'rels \times 'shared \Rightarrow \text{memory} \times 'p memref list \times 'owns \times 'rels \times 'shared \Rightarrow \text{bool}

\[ (- \rightarrow_w^* - [60,60] 100) \]

**where**

SBWrite_{sb}:

\[ (m, \text{Write}_{sb} \text{ volatile a sop v A L R W# rs, } \mathcal{O}, \mathcal{R}, S) \rightarrow_w (m(a := v), rs, \mathcal{O}, \mathcal{R}, S) \]

**abbreviation** store-buffer-steps::memory \times 'p store-buffer \times 'owns \times 'rels \times 'shared \Rightarrow \text{memory} \times 'p store-buffer \times 'owns \times 'rels \times 'shared \Rightarrow \text{bool}

\[ (- \rightarrow_w^* - [60,60] 100) \]

**where**

store-buffer-steps == store-buffer-step\(^*\)**

**term** x \(\rightarrow_w^* Y\)

**fun** buffered-val :: 'p memref list \Rightarrow \text{addr} \Rightarrow \text{val option}

**where**

buffered-val [] a = None
| buffered-val (r # rs) a' =
  (case r of
    Write_{sb} \text{ volatile a - v - - - - } \Rightarrow (\text{case buffered-val rs a'} of
      None \Rightarrow (\text{if a'}=a \text{ then Some v else None})
      | Some v' \Rightarrow \text{Some v'})
    | - \Rightarrow \text{buffered-val rs a'})

**definition** address-of :: 'p memref \Rightarrow \text{addr set}

**where**

address-of r = (case r of Write_{sb} \text{ volatile a - v - - - - } \Rightarrow \{a\} | Read_{sb} \text{ volatile a t v } \Rightarrow \{a\} | - \Rightarrow \{\})
lemma address-of-simps [simp]:
address-of (Write\textsubscript{sb} volatile a sop v A L R W) = \{a\}
address-of (Read\textsubscript{sb} volatile a t v) = \{a\}
address-of (Prog\textsubscript{sb} p p’ is) = \{
address-of (Ghost\textsubscript{sb} A L R W) = \{

(proof)

definition is-volatile :: ′p memref ⇒ bool
where
is-volatile r = (case r of Write\textsubscript{sb} volatile a - v - - - -⇒ volatile | Read\textsubscript{sb} volatile a t v ⇒ volatile
| - ⇒ False)

lemma is-volatile-simps [simp]:
is-volatile (Write\textsubscript{sb} volatile a sop v A L R W) = volatile
is-volatile (Read\textsubscript{sb} volatile a t v) = volatile
is-volatile (Prog\textsubscript{sb} p p’ is) = False
is-volatile (Ghost\textsubscript{sb} A L R W) = False

(proof)

definition is-Write\textsubscript{sb} :: ′p memref ⇒ bool
where
is-Write\textsubscript{sb} r = (case r of Write\textsubscript{sb} volatile a - v - - - -⇒ True | - ⇒ False)

definition is-Read\textsubscript{sb} :: ′p memref ⇒ bool
where
is-Read\textsubscript{sb} r = (case r of Read\textsubscript{sb} volatile a t v ⇒ True | - ⇒ False)

definition is-Prog\textsubscript{sb} :: ′p memref ⇒ bool
where
is-Prog\textsubscript{sb} r = (case r of Prog\textsubscript{sb} - - - -⇒ True | - ⇒ False)

definition is-Ghost\textsubscript{sb} :: ′p memref ⇒ bool
where
is-Ghost\textsubscript{sb} r = (case r of Ghost\textsubscript{sb} - - - -⇒ True | - ⇒ False)

lemma is-Write\textsubscript{sb}-simps [simp]:
is-Write\textsubscript{sb} (Write\textsubscript{sb} volatile a sop v A L R W) = True
is-Write\textsubscript{sb} (Read\textsubscript{sb} volatile a t v) = False
is-Write\textsubscript{sb} (Prog\textsubscript{sb} p p’ is) = False
is-Write\textsubscript{sb} (Ghost\textsubscript{sb} A L R W) = False

(proof)

lemma is-Read\textsubscript{sb}-simps [simp]:
is-Read\textsubscript{sb} (Read\textsubscript{sb} volatile a t v) = True
is-Read\textsubscript{sb} (Write\textsubscript{sb} volatile a sop v A L R W) = False
is-Read\textsubscript{sb} (Prog\textsubscript{sb} p p’ is) = False
is-Read\textsubscript{sb} (Ghost\textsubscript{sb} A L R W) = False

(proof)
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

(proof)

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

(proof)

definition is-volatile-Write\_sb:: \'p memref \Rightarrow 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

(proof)

lemma is-volatile-Write\_sb-address-of [simp]: is-volatile-Write\_sb x \implies address-of x \neq {}

(proof)

definition is-volatile-Read\_sb:: \'p memref \Rightarrow 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

(proof)

definition is-non-volatile-Write\_sb:: \'p memref \Rightarrow 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

(proof)
**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

(proof)

**lemma** is-volatile-split: is-volatile \(r\) =

(is-volatile-Read\(_{sb}\) \(r\) ∨ is-volatile-Write\(_{sb}\) \(r\))

(proof)

**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\(_{sb}\) \(r\))

(proof)

**fun** outstanding-refs:: (‘p memref ⇒ bool) ⇒ ‘p memref list ⇒ addr set

**where**

outstanding-refs \(P\) [] = {}

| outstanding-refs \(P\) (\(r\)#\(rs\)) = (if \(P\) \(r\) then (address-of \(r\)) ∪ (outstanding-refs \(P\) \(rs\))

else outstanding-refs \(P\) \(rs\))

**lemma** outstanding-refs-conv: outstanding-refs \(P\) \(sb\) = ∪ (address-of ′ \{r. \(r\) ∈ set \(sb\) ∧ \(P\) \(r\)\})

(proof)

**lemma** outstanding-refs-append:

∀ \(ys\). outstanding-refs vol (\(xs@ys\)) = outstanding-refs vol \(xs\) ∪ outstanding-refs vol \(ys\)

(proof)

**lemma** outstanding-refs-empty-negate: (outstanding-refs \(P\) \(sb\) = \{}) ⇒

(outstanding-refs (Not ◦ \(P\)) \(sb\) = ∪ (address-of ′ set \(sb\)))

(proof)

**lemma** outstanding-refs-mono-pred:

∀ \(sb\) \(sb′\).

∀ \(r. \(P\) \(r\) −→ \(P′\) \(r\) −impl outstanding-refs \(P\) \(sb\) ⊆ outstanding-refs \(P′\) \(sb′\)

(proof)

**lemma** outstanding-refs-mono-set:

∀ \(sb\) \(sb′\).

set \(sb\) ⊆ set \(sb′\) −impl outstanding-refs \(P\) \(sb\) ⊆ outstanding-refs \(P\) \(sb′\)

(proof)

52
lemma outstanding-refs-takeWhile:
outstanding-refs P (takeWhile P' sb) ⊆ outstanding-refs P sb
⟨proof⟩

lemma outstanding-refs-subsets:
outstanding-refs is-volatile-Write sb ⊆ outstanding-refs is-Write sb
outstanding-refs is-non-volatile-Write sb ⊆ outstanding-refs is-Write sb

outstanding-refs is-volatile-Read sb ⊆ outstanding-refs is-Read sb
outstanding-refs is-non-volatile-Read sb ⊆ outstanding-refs is-Read sb

outstanding-refs is-non-volatile-Write sb ⊆ outstanding-refs (Not ◦ is-volatile) sb
outstanding-refs is-non-volatile-Read sb ⊆ outstanding-refs (Not ◦ is-volatile) sb

outstanding-refs is-volatile-Write sb ⊆ outstanding-refs (is-volatile) sb
outstanding-refs is-volatile-Read sb ⊆ outstanding-refs (is-volatile) sb

outstanding-refs is-non-volatile-Write sb ⊆ outstanding-refs (Not ◦ is-volatile-Write sb)
outstanding-refs is-non-volatile-Read sb ⊆ outstanding-refs (Not ◦ is-volatile-Write sb)

outstanding-refs is-volatile-Write sb ⊆ outstanding-refs (is-volatile) sb
outstanding-refs is-volatile-Read sb ⊆ outstanding-refs (is-volatile) sb

⟨proof⟩

lemma outstanding-non-volatile-refs-conv:
outstanding-refs (Not ◦ is-volatile) sb = outstanding-refs is-non-volatile-Write sb ∪ outstanding-refs is-non-volatile-Read sb
⟨proof⟩

lemma outstanding-volatile-refs-conv:
outstanding-refs is-volatile sb = outstanding-refs is-volatile-Write sb ∪ outstanding-refs is-volatile-Read sb
⟨proof⟩

lemma outstanding-is-Write-ref-conv:
outstanding-refs is-Write sb = outstanding-refs is-non-volatile-Write sb ∪ outstanding-refs is-volatile-Write sb
⟨proof⟩

lemma outstanding-is-Read-ref-conv:
outstanding-refs is-Read sb = outstanding-refs is-non-volatile-Read sb ∪ outstanding-refs is-volatile-Read sb
⟨proof⟩

lemma outstanding-not-volatile-Read-ref-conv:
outstanding-refs (Not ◦ is-volatile-Read sb) =
outstanding-refs is-Write sb ∪ outstanding-refs is-non-volatile-Read sb

⟨proof⟩

lemmas misc-outstanding-refs-convs = outstanding-non-volatile-refs-conv
outstanding-volatile-refs-conv
outstanding-is-Write sb-ref-conv
outstanding-is-Read sb-ref-conv
outstanding-not-volatile-Read sb-ref-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)
⟨proof⟩

lemma outstanding-vol-write-takeWhile-append: outstanding-refs is-volatile-Write sb sb ≠ {} ⇒
   takeWhile (Not ◦ is-volatile-Write sb) (sb@xs) = (takeWhile (Not ◦ is-volatile-Write sb) sb) sb
⟨proof⟩

lemma no-outstanding-vol-write-dropWhile-append: outstanding-refs is-volatile-Write sb sb = {} ⇒
   dropWhile (Not ◦ is-volatile-Write sb) (sb@xs) = (dropWhile (Not ◦ is-volatile-Write sb) xs) sb@xs
⟨proof⟩

lemma outstanding-vol-write-dropWhile-append: outstanding-refs is-volatile-Write sb sb ≠ {} ⇒
   dropWhile (Not ◦ is-volatile-Write sb) (sb@xs) = (dropWhile (Not ◦ is-volatile-Write sb) sb)@xs
⟨proof⟩

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)
⟨proof⟩

lemma dropWhile-not-vol-write-empty: 
   outstanding-refs is-volatile-Write sb sb = {} ⇒ (dropWhile (Not ◦ is-volatile-Write sb) sb) = []
⟨proof⟩
**lemma** takeWhile-not-vol-write-outstanding-refs:
outstanding-refs is-volatile-Write\sb \(\text{takeWhile} \ (\text{Not} \ \circ \ \text{is-volatile-Write}\sb) \ \sb) = \{\}

(*proof*)

**lemma** no-volatile-Write\sb-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 \ True a' \text{sop}' v' A L R W))

(*proof*)

**lemma** no-volatile-Read\sb-conv: (outstanding-refs is-volatile-Read\sb \sb = \{\}) =
(\forall r \in \text{set} \ \sb. \ (\forall v' t' a'. \ r \neq \text{Read}\sb \ True t' v'))

(*proof*)

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

*where*

**SBReadBuffered:**
buffered-val \sb a = Some v
\implies
(Read volatile a t # is, sb, m, D, O, R, S) \rightarrow_{\sb} (is, \emptyset (t\mapsto), \sb, m, D, O, R, S)

**SBReadUnbuffered:**
buffered-val \sb a = None
\implies
(Read volatile a t # is, sb, m, D, O, R, S) \rightarrow_{\sb} (is, \emptyset (t\mapsto 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) \rightarrow_{\sb} (is, \emptyset, sb@[Write\sb False a (D, f) (f \emptyset) 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) \rightarrow_{\sb} (is, \emptyset, sb@[Write\sb True a (D, f) (f \emptyset) A L R W], m, D, O, R, S)

**SBFence:**
(Fence # is, \emptyset, [], m, D, O, R, S) \rightarrow_{\sb} (is, \emptyset, [], m, D, O, R, S)

**SBRMWReadOnly:**
\lnot \ \text{cond} (\emptyset(t\mapsto m a)) \implies
(RMW a t (D, f) \text{cond ret} A L R W# is, \emptyset, [], m, D, O, R, S) \rightarrow_{\sb} (is, \emptyset(t\mapsto m a), [], m, D, O, R, S)
SBRMWWrite:
[cond \((\delta(t\rightarrow m\ a))\)] \implies
\((\text{RMW} a\ t\ (D,f)\ \text{cond ret}\ A\ L\ R\ W\#\ \text{is},\ \emptyset,\ []\), m,D, O, R, S) \rightarrow_{sbh}
(is, \delta(t\rightarrow\text{ret}\ (m\ a)\ (f(\delta(t\rightarrow m\ a)))),[], m(a:= f(\delta(t\rightarrow m\ a))),D, O, R, S)

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

\text{inductive} \ sbh-memop-step ::
\[(\text{instrs} \times\ \text{tmpps} \times\ \text{'p store-buffer} \times\ \text{memory} \times\ \text{bool} \times\ \text{owns} \times\ \text{rels} \times\ \text{shared}) \implies
\]
\[(\text{instrs} \times\ \text{tmpps} \times\ \text{'p store-buffer} \times\ \text{memory} \times\ \text{bool} \times\ \text{owns} \times\ \text{rels} \times\ \text{shared}) \implies\ \text{bool}\]
\[
\text{where}
\]
SBHReadBuffered:
[buffered-val sb a = Some v]
\implies
\((\text{Read} \ \text{volatile} \ a\ t\ \#\ \text{is},\ \emptyset,\ sb, m, D, O, R, S) \rightarrow_{sbh}
\)
\[(is, \emptyset (t\rightarrow v), sb@[\text{Read}_{sb} \ \text{volatile} \ a\ t\ v], m, D, O, R, S)\]

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

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

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

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

SBHRMWReadOnly:
[\neg \ \text{cond} \ ((\delta(t\rightarrow m\ a))] \implies
\((\text{RMW} a\ t\ (D,f)\ \text{cond ret}\ A\ L\ R\ W\#\ \text{is},\ \emptyset,\ [], m, D, O, R, S) \rightarrow_{sbh} (is, \delta(t\rightarrow m\ a)\),[], m, False, O, \text{Map.empty}, S)

SBHRMWWrite:
[\text{cond} \ ((\delta(t\rightarrow 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_{sbh} (is, \delta(t\rightarrow\text{ret}\ (m\ a)\ (f(\delta(t\rightarrow m\ a)))),[], m(a:= f(\delta(t\rightarrow m\ a))), False, O \cup A \leftarrow R,\text{Map.empty}, S \oplus W\ R \ominus A\ L)\]

56
SBHGhost:
(\text{Ghost A L R W# is } \emptyset, \text{ sb, m, } D, O, R, S) \rightarrow_{\text{sbh}}
(is, \emptyset, \text{ sb@}[\text{Ghost sb} A L R W], m, D, O, R, S)

\textbf{interpretation direct}: memory-system direct-memop-step id-storebuffer-step \langle \text{proof} \rangle
\textbf{interpretation sb}: memory-system sb-memop-step store-buffer-step \langle \text{proof} \rangle
\textbf{interpretation sbh}: memory-system sbh-memop-step flush-step \langle \text{proof} \rangle

\textbf{primrec} non-volatile-owned-or-read-only:: \text{ bool } \Rightarrow \text{ shared } \Rightarrow \text{ owns } \Rightarrow \text{'a memref list } \Rightarrow \text{ bool }
\textbf{where}
non-volatile-owned-or-read-only pending-write \text{ S O [ ] } = \text{ True }
non-volatile-owned-or-read-only pending-write \text{ S O (x#xs) = } \text{(case x of Read sb volatile a t v } \Rightarrow \text{ (¬volatile } \rightarrow \text{ pending-write } \rightarrow \text{ (a } \in \text{ O } \lor \text{ a } \in \text{ read-only } S) } \land \text{ non-volatile-owned-or-read-only pending-write S O xs)}
\textbf{where}
Write sb volatile a sop v A L R W \Rightarrow \text{ (if volatile then non-volatile-owned-or-read-only True (S } \oplus \text{ W R } \ominus \text{ A L) (O } \cup \text{ A } \ominus \text{ R) xs else a } \in \text{ O } \land \text{ non-volatile-owned-or-read-only pending-write S O xs)}}
\textbf{primrec} acquired :: \text{ bool } \Rightarrow \text{'a memref list } \Rightarrow \text{ addr set } \Rightarrow \text{ addr set }
\textbf{where}
acquired pending-write [ ] A = (if pending-write then A else {})
acquired pending-write (x#xs) A = \text{(case x of Write sb volatile - - - A L R W } \Rightarrow \text{ (if volatile then acquired True xs (if pending-write then (A } \cup \text{ A’ } \ominus \text{ R) else (A’ } \ominus \text{ R)) else acquired pending-write xs A) Ghost sb A L R W } \Rightarrow \text{ acquired pending-write xs (if pending-write then (A } \cup \text{ A’ } \ominus \text{ R) else A) - } \Rightarrow \text{ acquired pending-write xs A})}
\textbf{primrec} share :: \text{'a memref list } \Rightarrow \text{ shared } \Rightarrow \text{ shared }
\textbf{where}
share [ ] S = S
share (x#xs) S = \text{(case x of Write sb volatile - - - A L R W } \Rightarrow \text{ (if volatile then (share xs (S } \oplus \text{ W R } \ominus \text{ A L)) else share xs S) Ghost sb A L R W } \Rightarrow \text{ share xs (S } \oplus \text{ W R } \ominus \text{ A L) - } \Rightarrow \text{ 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 \text{ volatile a t v}} ⇒
    (if pending-write ∧ ¬ volatile ∧ a ∈ A
    then insert a (acquired-reads pending-write xs A)
    else acquired-reads pending-write xs A)
  | Write_{sb \text{ 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
⟨proof⟩

lemma set-minus-mono-aux: A ⊆ B ⇒ A − C ⊆ B − C
⟨proof⟩

lemma acquired-mono: \( A, B \) pending-write. A ⊆ B ⇒ acquired pending-write xs A ⊆ acquired pending-write xs B
⟨proof⟩

lemma acquired-mono-in:
  assumes x-in: x ∈ acquired pending-write xs A
  assumes sub: A ⊆ B
  shows x ∈ acquired pending-write xs B
⟨proof⟩

lemma acquired-no-pending-write: A, B. acquired False xs A = acquired False xs B
⟨proof⟩

lemma acquired-no-pending-write-in:
  x ∈ acquired False xs A ⇒ x ∈ acquired False xs B
⟨proof⟩

lemma acquired-pending-write-mono-in: \( A, B \). x ∈ acquired False xs A ⇒ x ∈ acquired True xs B
⟨proof⟩

lemma acquired-pending-write-mono: acquired False xs A ⊆ acquired True xs B
⟨proof⟩

lemma acquired-append: \( A \) pending-write. acquired pending-write (xs@ys) A =
acquired (pending-write ∨ \text{outstanding-refs is-volatile-Write}_{sb} xs \neq \{\}) ys (acquired pending-write xs A)
⟨proof⟩

58
lemma acquired-take-drop:
  acquired (pending-write ∨ outstanding-refs is-volatile-Write\textsubscript{ab} (takeWhile P xs) \neq \{\})
  (dropWhile P xs) (acquired pending-write (takeWhile P xs) A) =
  acquired pending-write xs A
⟨proof⟩

lemma share-mono: \(\forall A B. \text{dom } A \subseteq \text{dom } B \implies \text{dom } (\text{share } xs \ A) \subseteq \text{dom } (\text{share } xs \ B)\)
⟨proof⟩

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 \(x \in \text{dom } (\text{share } xs \ B)\)
⟨proof⟩

lemma acquired-reads-mono:
  \(\forall A B \text{ pending-write}. \ A \subseteq B \implies \text{acquired-reads pending-write } xs \ A \subseteq \text{acquired-reads pending-write } xs \ B\)
⟨proof⟩

lemma acquired-reads-mono-in:
  assumes x-in: \(x \in \text{acquired-reads pending-write } xs \ A\)
  assumes sub: \(A \subseteq B\)
  shows \(x \in \text{acquired-reads pending-write } xs \ B\)
⟨proof⟩

lemma acquired-reads-no-pending-write: \(\forall A B. \text{acquired-reads False } xs \ A = \text{acquired-reads False } xs \ B\)
⟨proof⟩

lemma acquired-reads-no-pending-write-in:
  x \(\in \text{acquired-reads False } xs \ A \implies x \in \text{acquired-reads False } xs \ B\)
⟨proof⟩

lemma acquired-reads-pending-write-mono:
  \(\forall A. \text{acquired-reads False } xs \ A \subseteq \text{acquired-reads True } xs \ A\)
⟨proof⟩

lemma acquired-reads-pending-write-mono-in:
  assumes x-in: \(x \in \text{acquired-reads False } xs \ A\)
  shows \(x \in \text{acquired-reads True } xs \ A\)
⟨proof⟩

lemma acquired-reads-append: \(\bigwedge \text{pending-write } A. \text{acquired-reads pending-write } (xs@ys) \ A = \text{acquired-reads pending-write } xs \ A \cup \text{acquired-reads } (\text{pending-write } \lor \text{outstanding-refs is-volatile-Write}\textsubscript{ab} \ xs \neq \{\}) \ ys \ (\text{acquired pending-write } xs \ A)\)
⟨proof⟩
lemma in-acquired-reads-no-pending-write-outstanding-write:
\[ \forall A. \text{a} \in \text{acquired-reads} \text{False} \text{xs} \text{A} \implies \text{outstanding-refs (is-volatile-Write}_{ab} \text{xs} \neq \{\} \]

⟨proof⟩

lemma augment-read-only-mono: read-only \(S \subseteq \text{read-only } S'\) \implies
read-only \((S \oplus W) \subseteq \text{read-only } (S' \oplus W)\)

⟨proof⟩

lemma restrict-read-only-mono: read-only \(S \subseteq \text{read-only } S'\) \implies
read-only \((S \ominus A) \subseteq \text{read-only } (S' \ominus A)\)

⟨proof⟩

lemma share-read-only-mono:
\[ \forall S S'. \text{read-only } S \subseteq \text{read-only } S' \implies \]
read-only \((\text{share sb } S) \subseteq \text{read-only } (\text{share sb } S')\)

⟨proof⟩

lemma non-volatile-owned-or-read-only-append:
\[ \forall O S \text{ pending-write. non-volatile-owned-or-read-only pending-write } S O (xs@ys) = \]
\(\text{(non-volatile-owned-or-read-only pending-write } S O \text{xs } \land \)
\(\text{non-volatile-owned-or-read-only } (\text{pending-write } \lor \text{outstanding-refs is-volatile-Write}_{ab} \text{xs} \neq \{\})) \)
\((\text{share xs } S) \text{ (acquired True xs O) ys})

⟨proof⟩

lemma non-volatile-owned-or-read-only-mono:
\[ \forall O O' S \text{ pending-write. } O \subseteq O' \implies \text{non-volatile-owned-or-read-only pending-write } S O \]
x\(s \implies \text{non-volatile-owned-or-read-only pending-write } S O' \text{xs} \)

⟨proof⟩

lemma non-volatile-owned-or-read-only-shared-mono:
\[ \forall S S' O \text{ pending-write. } S \subseteq S' \implies \text{non-volatile-owned-or-read-only pending-write } S O \]
x\(s \implies \text{non-volatile-owned-or-read-only pending-write } S' O \text{xs} \)

⟨proof⟩

lemma non-volatile-owned-or-read-only-pending-write-antimono:
\[ \forall O S. \text{non-volatile-owned-or-read-only True } S O \text{xs} \implies \text{non-volatile-owned-or-read-only False } S O \text{xs} \]

⟨proof⟩

primrec all-acquired :: 'a memref list \(\Rightarrow\) addr set

where
\[ \text{all-acquired } \emptyset = \{\} \]
\(\mid \text{all-acquired } (i\#is) = \)
\((\text{case i of} \)

60
Write\textsubscript{sb}\ \text{volatile - - - A L R W} \Rightarrow (\text{if volatile then A } \cup \text{ all-acquired is else all-acquired is})

| \text{Ghost}_{sb} \ A L R W \Rightarrow A \cup \text{ all-acquired is}
| \Rightarrow \text{ all-acquired is})

\textbf{lemma} all-acquired-append: all-acquired (xs@ys) = all-acquired xs \cup all-acquired ys
\langle proof \rangle

\textbf{lemma} acquired-reads-all-acquired: \bigwedge O \text{ pending-write.}
acquired-reads pending-write sb O \subseteq O \cup \text{ all-acquired sb}
\langle proof \rangle

\textbf{lemma} acquired-takeWhile-non-volatile-Write\textsubscript{sb}:
\bigwedge A. (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) A) \subseteq A \cup \text{ all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb)}
\langle proof \rangle

\textbf{lemma} acquired-False-takeWhile-non-volatile-Write\textsubscript{sb}:
acquired False (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) A = \{\}
\langle proof \rangle

\textbf{lemma} outstanding-refs-takeWhile-opposite: outstanding-refs P (takeWhile (Not \circ P) xs) = \{\}
\langle proof \rangle

\textbf{lemma} no-outstanding-volatile-Write\textsubscript{sb}-acquired:
outstanding-refs is-volatile-Write\textsubscript{sb} sb = \{\} \implies acquired False sb A = \{\}
\langle proof \rangle

\textbf{lemma} acquired-all-acquired:\bigwedge pending-write A. acquired pending-write xs A \subseteq A \cup all-acquired xs
\langle proof \rangle

\textbf{lemma} acquired-all-acquired-in: x \in acquired pending-write xs A \implies x \in A \cup all-acquired xs
\langle proof \rangle

\textbf{primrec} sharing-consistent:: shared \Rightarrow owns \Rightarrow \{\text{a memref list} \Rightarrow \text{bool} \}
\textbf{where}
sharing-consistent \ S \ O \ || = \text{True}
| sharing-consistent \ S \ O \ (r\#rs) =
| \text{case r of}
| \text{Write}_{sb}\ \text{volatile - - - A L R W} \Rightarrow
| (\text{if volatile then} \ A \subseteq \text{dom} \ S \cup O \land L \subseteq \ A \land A \cap R = \{\} \land R \subseteq O \land
| \text{sharing-consistent} \ S \oplus_{R} \ominus_{A} L \ (O \cup A - R) \ rs
| \text{else sharing-consistent} \ S \ O \ rs)
lemma sharing-consistent-all-acquired:
\[ \forall \mathcal{O}. \text{sharing-consistent } \mathcal{S} \mathcal{O} \, \text{sb } \Rightarrow \text{all-acquired sb } \subseteq \text{dom } \mathcal{S} \cup \mathcal{O} \]
(proof)

lemma sharing-consistent-append:
\[ \forall \mathcal{S} \mathcal{O}. \text{sharing-consistent } \mathcal{S} \mathcal{O} (\text{xs@ys}) = \]
\[ (\text{sharing-consistent } \mathcal{S} \mathcal{O} \, \text{xs } \land \]
\[ \text{sharing-consistent } (\text{share xs } \mathcal{S}) (\text{acquired True xs } \mathcal{O}) \, \text{ys} \]
(proof)

primrec read-only-reads :: owns \Rightarrow t \text{ memref list } \Rightarrow \text{addr set}
where
read-only-reads \mathcal{O} [] = {}
| read-only-reads \mathcal{O} (x#xs) =
  (case x of
  Read_{\text{sb}} \text{ volatile a t v } \Rightarrow (\text{if } \neg \text{volatile } \land a \notin \mathcal{O} \)
  then insert a (read-only-reads \mathcal{O} xs)
  else read-only-reads \mathcal{O} xs)
| Write_{\text{sb}} \text{ volatile - - - A L R W } \Rightarrow
  (\text{if volatile then read-only-reads } (\mathcal{O} \cup A - R) \, \text{xs})
  else a \notin \text{dom } \mathcal{S} \land \text{read-only-reads } \mathcal{S} \mathcal{O} \, \text{xs})
| Ghost_{\text{sb}} A L R W \Rightarrow \text{read-only-reads } (\mathcal{O} \cup A - R) \, \text{xs}
| - \Rightarrow \text{read-only-reads } \mathcal{O} \, \text{xs})

lemma read-only-reads-append:
\[ \forall \mathcal{O}. \text{read-only-reads } \mathcal{O} (\text{xs@ys}) = \]
\[ \text{read-only-reads } \mathcal{O} \, \text{xs } \cup \text{read-only-reads } (\text{acquired True xs } \mathcal{O}) \, \text{ys} \]
(proof)

lemma read-only-reads-antimono:
\[ \forall \mathcal{O} \mathcal{O}'. \]
\[ \mathcal{O} \subseteq \mathcal{O}' \Rightarrow \text{read-only-reads } \mathcal{O}' \, \text{sb } \subseteq \text{read-only-reads } \mathcal{O} \, \text{sb} \]
(proof)

primrec non-volatile-writes-unshared:: shared \Rightarrow t \text{ a memref list } \Rightarrow \text{bool}
where
non-volatile-writes-unshared \mathcal{S} [] = True
| non-volatile-writes-unshared \mathcal{S} (x#xs) =
  (case x of
  Write_{\text{sb}} \text{ volatile a sop v A L R W } \Rightarrow (\text{if volatile then non-volatile-writes-unshared } (\mathcal{S}
\oplus_{\text{W}} R \ominus_{\text{A}} L) \text{xs})
  else a \notin \text{dom } \mathcal{S} \land \text{non-volatile-writes-unshared } \mathcal{S} \text{ xs})
| Ghost_{\text{sb}} A L R W \Rightarrow \text{non-volatile-writes-unshared } (\mathcal{S} \oplus_{\text{W}} R \ominus_{\text{A}} L) \, \text{xs}
| - \Rightarrow \text{non-volatile-writes-unshared } \mathcal{S} \, \text{xs} \]
**Lemma** non-volatile-writes-unshared-append:
\[ \forall S. \text{non-volatile-writes-unshared} S (xs @ ys) = (\text{non-volatile-writes-unshared} S \ {xs} \land \text{non-volatile-writes-unshared} (\text{share} \ {xs} S) \ {ys}) \]

(*proof*)

**Lemma** non-volatile-writes-unshared-antimono:
\[ \forall S S'. \text{dom} S \subseteq \text{dom} S' \implies \text{non-volatile-writes-unshared} S' \ {xs} \implies \text{non-volatile-writes-unshared} S \ {xs} \]

(*proof*)

**Primrec** no-write-to-read-only-memory:: shared \(\Rightarrow\) 'a memref list \(\Rightarrow\) bool

*where*

no-write-to-read-only-memory \(S\) [] = True

| no-write-to-read-only-memory \(S\) (x#xs) =

  (case x of
  Write\(_{sb}\) volatile \(a\) \(v\) A L R W \(\Rightarrow\) \(a\) \(\notin\) read-only \(S\) \land\)
  (if volatile then no-write-to-read-only-memory \((S \oplus W R \ominus A) L\) \(xs\)
  else no-write-to-read-only-memory \(S\) \(xs\))
  |
  Ghost\(_{sb}\) A L R W \(\Rightarrow\) no-write-to-read-only-memory \((S \oplus W R \ominus A) L\) \(xs\)
  |
  - \(\Rightarrow\) no-write-to-read-only-memory \(S\) \(xs\))

**Lemma** no-write-to-read-only-memory-append:
\[ \forall S. \text{no-write-to-read-only-memory} S (xs @ ys) = (\text{no-write-to-read-only-memory} S \ {xs} \land \text{no-write-to-read-only-memory} (\text{share} \ {xs} S) \ {ys}) \]

(*proof*)

**Lemma** no-write-to-read-only-memory-antimono:
\[ \forall S S'. S \subseteq S' \implies \text{no-write-to-read-only-memory} S' \ {xs} \implies \text{no-write-to-read-only-memory} S \ {xs} \]

(*proof*)

**Locale** outstanding-non-volatile-refs-owned-or-read-only =

*fixes* \(S::\text{shared}\)

*assumes* outstanding-non-volatile-refs-owned-or-read-only:
\[ \forall i \text{is} O R D \ {\theta} sb p. \]
\[ [i < \text{length ts}; ts!i = (p,\text{is},\theta,\text{sb},D,O,R)] \]
\[ \implies \neg \text{non-volatile-owned-or-read-only} \ {\text{False}} S O sb \]

**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:
\[ \forall i p_i \text{is}_i O_i R_i D_i \ {\theta}_i sb_i j p_j \text{is}_j O_j R_j D_j \ {\theta}_j sb_j. \]
\[ [i < \text{length ts}; j < \text{length ts}; i \neq j]; \]
\[ ts!i = (p_i,\text{is}_i,\theta_i,\text{sb}_i,D_i,O_i,R_i); ts!j = (p_j,\text{is}_j,\theta_j,\text{sb}_j,D_j,O_j,R_j) \]
\]
\[ (O_j \cup \text{all-acquired } s_b) \cap \text{outstanding-refs is-volatile-Write}_a \text{ sb}_i = \{\} \]

**locale** read-only-reads-unowned =

**fixes** ts::'(p, p store-buffer, bool, owns, rels) thread-config list

**assumes** read-only-reads-unowned:
\[
\forall i \ j \ p \ i \ p \ j \ O_i \ R_i \ D_i \ \hat{O}_i \ \hat{sb}_i \ j \ p \ j \ i \ j \ O_j \ R_j \ D_j \ \hat{O}_j \ \hat{sb}_j.
\]
\[
[i < \text{length ts}; j < \text{length ts}; i \neq j;\]
\[\text{ts}_i = (p_i, i, \hat{O}_i, \hat{sb}_i, D_i, O_i, R_i); \text{ts}_j = (p_j, j, \hat{O}_j, \hat{sb}_j, D_j, O_j, R_j)\]
\[\implies (O_i \cup \text{all-acquired } s_b) \cap \text{read-only-reads (acquired True (takeWhile (Not \circ \text{is-volatile-Write}_a \text{ sb}_i) \text{ sb}_i) O_i)) }\]
\[
\text{(dropWhile (Not \circ \text{is-volatile-Write}_a \text{ sb}_i) \text{ sb}_i) = \{\}}
\]

**locale** ownership-distinct =

**fixes** ts::'(p, p store-buffer, bool, owns, rels) thread-config list

**assumes** ownership-distinct:
\[
\forall i \ j \ p \ i \ p \ j \ O_i \ R_i \ D_i \ \hat{O}_i \ \hat{sb}_i \ j \ p \ j \ i \ j \ O_j \ R_j \ D_j \ \hat{O}_j \ \hat{sb}_j.
\]
\[
[i < \text{length ts}; j < \text{length ts}; i \neq j;\]
\[\text{ts}_i = (p_i, i, \hat{O}_i, \hat{sb}_i, D_i, O_i, R_i); \text{ts}_j = (p_j, j, \hat{O}_j, \hat{sb}_j, D_j, O_j, R_j)\]
\[\implies (O_i \cup \text{all-acquired } s_b) \cap (O_j \cup \text{all-acquired } s_b) = \{\}
\]

**locale** valid-ownership =

- outstanding-non-volatile-refs-owned-or-read-only +
- outstanding-volatile-writes-unowned-by-others +
- read-only-reads-unowned +
- ownership-distinct

**locale** outstanding-non-volatile-writes-unshared =

**fixes** S::shared and ts::'(p, p store-buffer, bool, owns, rels) thread-config list

**assumes** outstanding-non-volatile-writes-unshared:
\[
\forall i \ p \ i \ p \ j \ O \ R \ D \ \hat{O} \ \hat{sb}.
\]
\[
[i < \text{length ts}; \text{ts}_i = (p, i, \hat{O}, \hat{sb}, D, O, R) ]
\]
\[\implies \text{non-volatile-writes-unshared } S \text{ sb}
\]

**locale** sharing-consis =

**fixes** S::shared and ts::'(p, p store-buffer, bool, owns, rels) thread-config list

**assumes** sharing-consis:
\[
\forall i \ p \ i \ p \ j \ O \ R \ D \ \hat{O} \ \hat{sb}.
\]
\[
[i < \text{length ts}; \text{ts}_i = (p, i, \hat{O}, \hat{sb}, D, O, R) ]
\]
\[\implies \text{sharing-consistent } S \ O \ \text{ sb}
\]
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:
\[ \forall i. \ p \ is \ O \ R \ D \ \emptyset \ sb. \]
\[ [i < \ length \ ts; \ ts!i \ = \ (p, is, \emptyset, sb, D, O, R) ] \]
\[ \implies \]
no-write-to-read-only-memory $S$ sb

locale valid-sharing =
outstanding-non-volatile-writes-unshared +
sharing-consis +
read-only-unowned +
unowned-shared +
no-outstanding-write-to-read-only-memory

locale valid-ownership-and-sharing = valid-ownership +
outstanding-non-volatile-writes-unshared +
sharing-consis +
no-outstanding-write-to-read-only-memory

lemma (in read-only-reads-unowned)
read-only-reads-unowned-nth-update:
\[ \forall i. \ p \ is \ O \ R \ D \ \emptyset \ sb. \]
\[ [i < \ length \ ts; \ ts!i \ = \ (p, is, \emptyset, sb, D, O, R) ] \]
read-only-reads (acquired True (takeWhile (Not \circ is-volatile-Write sb) sb') O')
\[ (\dropWhile (Not \circ is-volatile-Write sb) sb') \subseteq \read-only-reads \ (acquired \ True \) \]
\[ (\takeWhile (Not \circ is-volatile-Write sb) sb) O) \]
\[ (\dropWhile (Not \circ is-volatile-Write sb) sb); \]
\[ O' \cup \all-acquired \ sb' \subseteq O \cup \all-acquired \ sb \]
\[ \implies \]
read-only-reads-unowned (ts[i := (p', is', \emptyset', sb', D', O', R')])
(proof)

lemma outstanding-non-volatile-refs-owned-or-read-only-tl:
outstanding-non-volatile-refs-owned-or-read-only-tl $S$ (t#ts) \[ \implies \]
outstanding-non-volatile-refs-owned-or-read-only $S$ ts
(proof)

lemma outstanding-volatile-writes-unowned-by-others-tl:
outstanding-volatile-writes-unowned-by-others-tl $S$ (t#ts) \[ \implies \]
outstanding-volatile-writes-unowned-by-others ts
(proof)

lemma read-only-reads-unowned-tl:
read-only-reads-unowned-tl $S$ (t # ts) \[ \implies \]
read-only-reads-unowned (ts)

(proof)

**Lemma ownership-distinct-tl:**
**Assumes** dist: ownership-distinct (t#ts)
**Shows** ownership-distinct ts
(proof)

**Lemma valid-ownership-tl:** valid-ownership $S$ (t#ts) $\implies$ valid-ownership $S$ ts
(proof)

**Lemma sharing-consistent-takeWhile:**
**Assumes** consis: sharing-consistent $S$ $O$ sb
**Shows** sharing-consistent $S$ $O$ (takeWhile $P$ sb)
(proof)

**Lemma sharing-consis-tl:** sharing-consis $S$ (t#ts) $\implies$ sharing-consis $S$ ts
(proof)

**Lemma sharing-consis-Cons:**

[sharing-consis $S$ ts; sharing-consistent $S$ $O$ sb] $\implies$ sharing-consis $S$ ((p,is,0,sl,O,R)#ts)
(proof)

**Lemma outstanding-non-volatile-writes-unshared-tl:**
outstanding-non-volatile-writes-unshared $S$ (t#ts) $\implies$ outstanding-non-volatile-writes-unshared $S$ ts
(proof)

**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
(proof)

**Lemma valid-ownership-and-sharing-tl:**
valid-ownership-and-sharing $S$ (t#ts) $\implies$ valid-ownership-and-sharing $S$ ts
(proof)

**Lemma non-volatile-owned-or-read-only-outstanding-non-volatile-writes:**

$\forall S$ pending-write. [non-volatile-owned-or-read-only pending-write $S$ $O$ sb] $\implies$ outstanding-refs is-non-volatile-Write$_{sb}$ sb $\subseteq$ $O$ all-acquired sb
(proof)

**Lemma** (in outstanding-non-volatile-refs-owned-or-read-only)
outstanding-non-volatile-writes-owned:
\textbf{assumes} i-bound: \(i < \text{length } ts\)
\textbf{assumes} \(ts[i]= (p, \text{is}, s, \text{sb}, D, O, R)\)
\textbf{shows} outstanding-refs is-non-volatile-Write\(_{sb}\) \(sb \subseteq O \cup \text{all-acquired } sb\)
\textit{proof}

\textbf{lemma} non-volatile-reads-acquired-or-read-only:
\[
\forall O S. \text{[non-volatile-owned-or-read-only } \text{True } S O sb; \text{sharing-consistent } S O sb]\]
\[
\Rightarrow
\text{outstanding-refs is-non-volatile-Read}_{sb} \text{ sb } \subseteq O \cup \text{all-acquired } sb \cup \text{read-only } S
\]
\textit{proof}

\textbf{lemma} non-volatile-reads-acquired-or-read-only-reads:
\[
\forall O S \text{ pending-write. } \text{[non-volatile-owned-or-read-only pending-write } S O sb]\]
\[
\Rightarrow
\text{outstanding-refs is-non-volatile-Read}_{sb} \text{ sb } \subseteq O \cup \text{all-acquired } sb \cup \text{read-only-reads } O sb
\]
\textit{proof}

\textbf{lemma} non-volatile-owned-or-read-only-outstanding-refs:
\[
\forall O S \text{ pending-write. } \text{[non-volatile-owned-or-read-only pending-write } S O sb]\]
\[
\Rightarrow
\text{outstanding-refs (Not } \circ \text{ is-volatile) sb } \subseteq O \cup \text{all-acquired sb } \cup \text{read-only-reads } O sb
\]
\textit{proof}

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

\textbf{lemma} read-only-reads-read-only:
\[
\forall S O. \text{[non-volatile-owned-or-read-only } \text{True } S O sb; \text{sharing-consistent } S O sb]\]
\[
\Rightarrow
\text{read-only-reads } O sb \subseteq O \cup \text{all-acquired sb } \cup \text{read-only } S
\]
\textit{proof}

\textbf{lemma} no-unacquired-write-to-read-only-reads:
\[
\forall S O. \text{[no-write-to-read-only-memory } S sb;\]
\[
\Rightarrow
\]
non-volatile-owned-or-read-only True $S \ O \ sb$; sharing-consistent $S \ O \ sb$

$a \in \text{read-only-reads} \ O \ sb$; $a \notin \ (O \cup \text{all-acquired} \ sb)$

$$\implies a \notin \text{outstanding-refs is-Write}_{sb} \ sb$$

(proof)

lemma no-unacquired-write-to-read-only:

assumes no-wrt: no-write-to-read-only-memory $S \ sb$

assumes consis: sharing-consistent $S \ O \ sb$

shows read-only $S \cap \text{outstanding-refs is-Write}_{sb} \ sb \subseteq O \cup \text{all-acquired} \ sb$

(proof)

lemma no-unacquired-volatile-write-to-read-only:

assumes no-wrt: no-write-to-read-only-memory $S \ sb$

assumes consis: sharing-consistent $S \ O \ sb$

shows read-only $S \cap \text{outstanding-refs is-volatile-Write}_{sb} \ sb \subseteq O \cup \text{all-acquired} \ sb$

(proof)

lemma no-unacquired-non-volatile-write-to-read-only:

assumes no-wrt: no-write-to-read-only-memory $S \ sb$

assumes consis: sharing-consistent $S \ O \ sb$

shows read-only-reads $O \sb \cap \text{outstanding-refs is-non-volatile-Write}_{sb} \ sb \subseteq O \cup \text{all-acquired} \ sb$

(proof)

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 $O \sb \cap \text{outstanding-refs is-Write}_{sb} \ sb \subseteq O \cup \text{all-acquired} \ sb$

(proof)

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 $O \sb \cap \text{outstanding-refs is-volatile-Write}_{sb} \ sb \subseteq O \cup \text{all-acquired} \ sb$

(proof)

lemma no-unacquired-non-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 $O \sb \cap \text{outstanding-refs is-non-volatile-Write}_{sb} \ sb \subseteq O \cup \text{all-acquired} \ sb$

(proof)
lemma set-dropWhileD: \( x \in \text{set} \ (\text{dropWhile} \ P \ \text{x}s) \implies x \in \text{set} \ \text{x}s \)
\((proof)\)

lemma outstanding-refs-takeWhileD:
\( x \in \text{outstanding-refs} \ P \ (\text{takeWhile} \ P' \ \text{sb}) \implies x \in \text{outstanding-refs} \ P \ \text{sb} \)
\((proof)\)

lemma outstanding-refs-dropWhileD:
\( x \in \text{outstanding-refs} \ P \ (\text{dropWhile} \ P' \ \text{sb}) \implies x \in \text{outstanding-refs} \ P \ \text{sb} \)
\((proof)\)

lemma dropWhile-ConsD: dropWhile \ P \ \text{x}s = y#y\text{s} \implies \neg P \ y
\((proof)\)

lemma non-volatile-owned-or-read-only-drop:
non-volatile-owned-or-read-only False \( S \ O \ \text{sb} \)
\(\implies\) non-volatile-owned-or-read-only True
\((share \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ \text{S})\)
\((acquired \ True \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ \text{O})\)
\((\text{dropWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb})\)
\((proof)\)

lemma read-only-share: \( \bigwedge S \ \text{O}. \)
sharing-consistent \( S \ \text{O} \ \text{sb} \implies\)
\(\text{read-only} \ (\text{share} \ \text{sb} \ \text{S}) \subseteq \text{read-only} \ \text{S} \cup \text{O} \cup \text{all-acquired} \ \text{sb}\)
\((proof)\)

lemma (in valid-ownership-and-sharing) outstanding-non-write-non-vol-reads-drop-disj:
assumes i-bound: \( i < \text{length} \ \text{ts} \)
assumes j-bound: \( j < \text{length} \ \text{ts} \)
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 outstanding-refs is-Write_{sb} (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) \ text{sb}_i) \cap
\text{outstanding-refs} is-non-volatile-Read_{sb} (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) \ \text{sb}_j)
= \{\} \)
\((proof)\)

lemma (in valid-ownership-and-sharing) outstanding-non-volatile-write-disj:
assumes i-bound: \( i < \text{length} \ \text{ts} \)
assumes j-bound: \( j < \text{length} \ \text{ts} \)
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 outstanding-refs (is-non-volatile-Write_{sb}) (takeWhile (Not ∘ is-volatile-Write_{sb}) sb_i) ∩
  (outstanding-refs is-volatile-Write_{sb} sb_j) ∪
  outstanding-refs is-non-volatile-Write_{sb} sb_j ∪
  outstanding-refs is-non-volatile-Read_{sb} (dropWhile (Not ∘ is-volatile-Write_{sb}) sb_j)
∪
  (outstanding-refs is-non-volatile-Read_{sb} (takeWhile (Not ∘ is-volatile-Write_{sb}) sb_j) −
    read-only-reads O_j (takeWhile (Not ∘ is-volatile-Write_{sb}) sb_j)) ∪
  (O_j ∪ all-acquired (takeWhile (Not ∘ is-volatile-Write_{sb}) sb_j))
) = {} (is ?non-vol-writes-i ∩ ?not-volatile-j = {})
⟨proof⟩

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, θ_i, sb_i, D_i, O_i, R_i)
assumes jth: ts!j = (p_j, is_j, θ_j, sb_j, D_j, O_j, R_j)
shows outstanding-refs (is-non-volatile-Write_{sb}) (takeWhile (Not ∘ is-volatile-Write_{sb}) sb_j)
  ∩
  outstanding-refs (Not ∘ is-volatile-Read_{sb}) (dropWhile (Not ∘ is-volatile-Write_{sb}) sb_j) = {}
  (is ?non-vol-writes-i ∩ ?not-volatile-j = {})
⟨proof⟩

lemma (in valid-ownership-and-sharing) outstanding-refs-is-Write_{sb}-takeWhile-disj:
 ∀ i < length ts. (forall j < length ts. i ̸= j →
  (let (r,r,sb,r,r) = tsli;
       r,r,sb,r,r = tslj
  in outstanding-refs is-Write_{sb} sb_i ∩
    outstanding-refs is-Write_{sb} (takeWhile (Not ∘ is-volatile-Write_{sb}) sb_j) =
    {}))
⟨proof⟩

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 ∈ read-tmps xs) = (∃ volatile a v. Read_{sb} volatile a t v ∈ set xs)
lemma read-tmps-mono: $\forall ys. \text{set } xs \subseteq \text{set } ys \implies \text{read-tmps } xs \subseteq \text{read-tmps } ys$

(fun distinct-read-tmps:: 'p store-buffer ⇒ bool
where
  distinct-read-tmps [] = True
| distinct-read-tmps (r#rs) =
    (case r of
      Read$_{ab}$ volatile a t v ⇒ t $\not\in$ (read-tmps rs) $\land$ distinct-read-tmps rs
      | - ⇒ distinct-read-tmps rs)

lemma distinct-read-tmps-conv:
  distinct-read-tmps xs = ($\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j \implies$
    (case xs!i of
      Read$_{ab} \Rightarrow$ case xs!j of Read$_{ab} \Rightarrow t_i \neq t_j | - ⇒ True)
| - ⇒ True))

 — Nice lemma, ugly proof.

(proof)

(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 $\in$ load-tmps xs) = ($\exists$ volatile a. Read volatile a t $\in$ set xs) $\lor$
    ($\exists$ a sop cond ret A L R W. RMW a t sop cond ret A L R W $\in$ set xs))

(proof)

lemma load-tmps-mono: $\forall ys. \text{set } xs \subseteq \text{set } ys \implies \text{load-tmps } xs \subseteq \text{load-tmps } ys$

(fun distinct-load-tmps:: instrs ⇒ bool
where
  distinct-load-tmps [] = True
| distinct-load-tmps (r#rs) =
    (case r of
      Read volatile a t ⇒ t $\not\in$ (load-tmps rs) $\land$ distinct-load-tmps rs
      | RMW a t sop cond ret A L R W ⇒ t $\not\in$ (load-tmps rs) $\land$ distinct-load-tmps rs
      | - ⇒ distinct-load-tmps rs)
locale load-tmps-distinct =
fixes ts::\(\langle p, p \text{ store-buffer, bool, owns, rels} \rangle\) thread-config list
assumes load-tmps-distinct:
\[
\forall i \ p \ \text{is} \ \mathcal{O} \mathcal{R} \mathcal{D} \neq \emptyset \ \text{sb}.
\[i < \text{length ts}; \ ts!i = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R}) \]\]
\[
\implies \ \text{distinct-load-tmps is}
\]

locale read-tmps-distinct =
fixes ts::\(\langle p, p \text{ store-buffer, bool, owns, rels} \rangle\) thread-config list
assumes read-tmps-distinct:
\[
\forall i \ p \ \text{is} \ \mathcal{O} \mathcal{R} \mathcal{D} \neq \emptyset \ \text{sb}.
\[i < \text{length ts}; \ ts!i = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R}) \]\]
\[
\implies \ \text{distinct-read-tmps sb}
\]

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

locale tmps-distinct =
load-tmps-distinct +
read-tmps-distinct +
load-tmps-read-tmps-distinct

lemma rev-read-tmps: \text{read-tmps} (\text{rev} \ xs) = \text{read-tmps} xs
\langle proof \rangle

lemma rev-load-tmps: \text{load-tmps} (\text{rev} \ xs) = \text{load-tmps} xs
\langle proof \rangle

lemma distinct-read-tmps-append:\ \forall ys. \ \text{distinct-read-tmps} (xs @ ys) =
\ (\text{distinct-read-tmps} xs \ \land \ \text{distinct-read-tmps} ys \ \land \ \text{read-tmps} xs \ \cap \ \text{read-tmps} ys = \{\})
\langle proof \rangle

lemma distinct-load-tmps-append:\ \forall ys. \ \text{distinct-load-tmps} (xs @ ys) =
\ (\text{distinct-load-tmps} xs \ \land \ \text{distinct-load-tmps} ys \ \land \ \text{load-tmps} xs \ \cap \ \text{load-tmps} ys = \{\})
\langle proof \rangle

lemma read-tmps-append: \text{read-tmps} (xs@ys) = (\text{read-tmps} xs \ \cup \ \text{read-tmps} ys)
\langle proof \rangle

lemma load-tmps-append: \text{load-tmps} (xs@ys) = (\text{load-tmps} xs \ \cup \ \text{load-tmps} ys)
fun write-sops:: \('p store-buffer \Rightarrow \text{sop set}
where
write-sops [] = {}
| write-sops (r#rs) =
  (case r of
   Write sb volatile a sop v - - - -\Rightarrow insert sop (write-sops rs)
  | - \Rightarrow write-sops rs)

lemma in-write-sops-conv:
(sop \in write-sops xs) = (\exists \text{volatile a v A L R W. Write}_{sb} \text{volatile a sop v A L R W} \in \text{set xs})
(proof)

lemma write-sops-mono: \(\forall ys. \text{set xs} \subseteq \text{set ys} \Rightarrow \text{write-sops xs} \subseteq \text{write-sops ys}
(proof)

lemma write-sops-append: write-sops (xs@ys) = write-sops xs \cup write-sops ys
(proof)

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

lemma in-store-sops-conv:
(sop \in store-sops xs) = ((\exists \text{volatile a v A L R W. Write}_{volatile} \text{a sop v A L R W} \in \text{set xs})
\lor
(\exists a t \text{cond ret A L R W. RMW}_{a t} \text{sop cond ret A L R W} \in \text{set xs}))
(proof)

lemma store-sops-mono: \(\forall ys. \text{set xs} \subseteq \text{set ys} \Rightarrow \text{store-sops xs} \subseteq \text{store-sops ys}
(proof)

lemma store-sops-append: store-sops (xs@ys) = store-sops xs \cup store-sops ys
(proof)

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

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

locale valid-sops = valid-write-sops + valid-store-sops
The value stored in a non-volatile 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 owns \Rightarrow memory \Rightarrow 'p store-buffer \Rightarrow bool

where
reads-consistent pending-write \mathcal{O} m [] = True
| reads-consistent pending-write \mathcal{O} m (r#rs) =
  (case r of
   Read_{sb} volatile a t v \Rightarrow (\neg \text{volatile} \rightarrow (\text{pending-write} \lor a \in \mathcal{O}) \rightarrow v = m a) \land
   \text{reads-consistent pending-write} \mathcal{O} m rs
| Write_{sb} volatile a sop v A L R W \Rightarrow
  \begin{cases}
    \text{if volatile then} \\
    \text{outstandingRefs is-volatile-Read_{sb} rs} = \{\} \land
    \text{reads-consistent True (O U A - R) (m(a := v)) rs} \\
    \text{else reads-consistent pending-write} \mathcal{O} (m(a := v)) rs
  \end{cases}
| Ghost_{sb} A L R W \Rightarrow \text{reads-consistent pending-write} (\mathcal{O} U A - R) m rs
| - \Rightarrow \text{reads-consistent pending-write} \mathcal{O} m rs
} )

fun volatile-reads-consistent:: memory \Rightarrow 'p store-buffer \Rightarrow bool

where
volatile-reads-consistent m [] = True
| volatile-reads-consistent m (r#rs) =
  (case r of
   Read_{sb} volatile a t v \Rightarrow (\text{volatile} \rightarrow v = m a) \land \text{volatile-reads-consistent m rs}
| Write_{sb} volatile a sop v A L R W \Rightarrow \text{volatile-reads-consistent (m(a := v)) rs}
| - \Rightarrow \text{volatile-reads-consistent m rs}
} )

fun flush:: 'p store-buffer \Rightarrow memory \Rightarrow memory

where
flush [] m = m
| flush (r#rs) m =
  (case r of
   Write_{sb} volatile a - v - - - - \Rightarrow \text{flush rs (m(a := v))}
| - \Rightarrow \text{flush rs m})

lemma reads-consistent-pending-write-antimono:
\[ \mathcal{O} m. \text{reads-consistent True } \mathcal{O} m \text{ sb } \Rightarrow \text{reads-consistent False } \mathcal{O} m \text{ sb} \]
lemma reads-consistent-owns-antimono:
\[\forall O O' \text{ pending-write } m.\]
\[O \subseteq O' \implies \text{reads-consistent pending-write } O' m \text{ sb } \implies \text{reads-consistent pending-write } O m \text{ sb}\]
⟨proof⟩

lemma acquired-reads-mono': \(x \in \text{acquired-reads } b \text{ xs } A \implies \text{acquired-reads } b \text{ xs } B = \{\}\)
\[\implies A \subseteq B \implies \text{False}\]
⟨proof⟩

lemma reads-consistent-append:
\[\forall m \text{ pending-write } O. \text{reads-consistent pending-write } O m (xs@ys) = \]
\[(\text{reads-consistent pending-write } O m \text{ xs } \land \]
\[\text{reads-consistent (pending-write } \lor \text{ outstanding-ref is-volatile-Write}_{sb} \text{ xs } \neq \{\}) \]
\[(\text{acquired True xs } O) \text{ (flush xs m) } ys \land \]
\[(\text{outstanding-ref is-volatile-Write}_{sb} \text{ xs } \neq \{\}) \]
\[\implies \text{outstanding-ref is-volatile-Read}_{sb} \text{ ys } = \{\})\]
⟨proof⟩

lemma reads-consistent-mem-eq-on-non-volatile-reads:
assumes mem-eq: \(\forall a \in A. m' a = m a\)
assumes subset: outstanding-ref (is-non-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: reads-consistent pending-write O m sb
shows reads-consistent pending-write O m' sb
⟨proof⟩

lemma volatile-reads-consistent-mem-eq-on-volatile-reads:
assumes mem-eq: \(\forall a \in A. m' a = m a\)
assumes subset: outstanding-ref (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
⟨proof⟩

locale valid-reads =
fixes m::memory and ts::(\'p, \'p store-buffer, bool, owns, rels) thread-config list
assumes valid-reads: \(\forall i p \in O R D \emptyset \text{ sb}.\)
\[\[i < \text{length ts}; ts!i = (p, is, \emptyset, sb, D, O, R) \] \implies \]
\[\text{reads-consistent False } O m \text{ sb}\]
lemma valid-reads-Cons: valid-reads m (t#ts) =

75
\[
\text{let } (\text{-,-,-,sb},O,) = t \text{ in reads-consistent False } O \text{ m sb } \land \text{ valid-reads m ts) }
\]

\textbf{proof}

\text{Reads}_{sb} \text{ and writes have in the store-buffer have to conform to the valuation of temporaries.}

\textbf{context} program

\textbf{begin}

\textbf{fun} history-consistent:: \text{tmps} \Rightarrow \text{'p} \Rightarrow \text{'p} \text{ store-buffer} \Rightarrow \text{bool}

\textbf{where}

history-consistent \text{\hat{\theta}} \text{ p} [] = \text{True}

| history-consistent \text{\hat{\theta}} \text{ p} (r#rs) =

\text{(case } r \text{ of}

| \text{Read}_{sb} \text{ vol a t v } \Rightarrow

\text{(case } \text{\hat{\theta}} \text{ t of Some } v' \Rightarrow v = v' \land \text{history-consistent } \text{\hat{\theta}} \text{ p} \text{ rs } | - \Rightarrow \text{False)}

| \text{Write}_{sb} \text{ vol a (D,f)} v - - - - \Rightarrow

D \subseteq \text{dom } \text{\hat{\theta}} \land f \text{\hat{\theta}} = v \land D \cap \text{read-tmps rs} = \{\} \land \text{history-consistent } \text{\hat{\theta}} \text{ p} \text{ rs}

| - \Rightarrow \text{history-consistent } \text{\hat{\theta}} \text{ p} \text{ rs)

\textbf{end}

\textbf{fun} hd-prog:: \text{'p} \Rightarrow \text{'p} \text{ store-buffer} \Rightarrow \text{'p}

\textbf{where}

hd-prog \text{ p} [] = \text{p}

| hd-prog \text{ p} (i#is) = \text{(case } i \text{ of}

| Prog}_{sb} \text{ p}_1 \text{ p}_2 \text{ is } \Rightarrow \text{p}_1 = \text{p} \land

\text{\hat{\theta}}(\text{dom } \text{\hat{\theta}} - \text{read-tmps rs}) \vdash \text{p}_1 \rightarrow \text{p} \text{ (p}_2,\text{is}) \land

\text{history-consistent } \text{\hat{\theta}} \text{ p} \text{ rs)

\textbf{end}

\textbf{fun} last-prog:: \text{'p} \Rightarrow \text{'p} \text{ store-buffer} \Rightarrow \text{'p}

\textbf{where}

last-prog \text{ p} [] = \text{p}

| last-prog \text{ p} (i#is) = \text{(case } i \text{ of}

| Prog}_{sb} \text{ - p'} \text{ - } \Rightarrow \text{last-prog p'} \text{ is}

| - \Rightarrow \text{last-prog p} \text{ is)

\textbf{locale} valid-history = program +

\textbf{constrains}

\text{program-step ::} \text{tmps} \Rightarrow \text{'p} \Rightarrow \text{'p} \times \text{instrs} \Rightarrow \text{bool}

\textbf{fixes} ts::(\text{'p},\text{'p} \text{ store-buffer,bool,owns,rels}) \text{ thread-config list}

\textbf{assumes}

\text{valid-history:} \bigwedge i \text{ p} \in O \ R \ D \ \text{\hat{\theta}} \text{ sb.}

[i < \text{length ts; ts}[i] = (p,\text{is,\hat{\theta},sb,D,O,R}) ] \implies

\text{program.history-consistent program-step } \text{\hat{\theta}} (\text{hd-prog p sb} \text{ sb)

\textbf{fun} data-dependency-consistent-instrs:: \text{addr set} \Rightarrow \text{instrs} \Rightarrow \text{bool}

\textbf{where}

data-dependency-consistent-instrs \text{T} [] = \text{True}

| data-dependency-consistent-instrs \text{T} (i#is) =

\text{(case } i \text{ of}

| Write volatile a (D,f) - - - - \Rightarrow D \subseteq T \land D \cap \text{load-tmps is} = \{\} \land

data-dependency-consistent-instrs \text{T} \text{ is

76
\[ \text{RMW} \ a \ t \ (D,f) \ \text{cond} \ \text{ret} \ - \ - \ - \Rightarrow D \subseteq T \land D \cap \text{load-tmps} = \{} \ \land \ \text{data-dependency-consistent-instrs} \ (\text{insert} \ t \ T) \ \text{is} \]

| - \Rightarrow \text{data-dependency-consistent-instrs} \ (\text{insert} \ t \ T) \ \text{is} 
| - \Rightarrow \text{data-dependency-consistent-instrs} \ T \ \text{is} 

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

\textit{proof}

\textbf{lemma} \ \text{data-dependency-consistent-instrs-append}:
\[ \forall ys T. \ \text{data-dependency-consistent-instrs} \ T \ (xs@ys) = \]
\[ (\text{data-dependency-consistent-instrs} \ T \ xs \land \text{data-dependency-consistent-instrs} \ (T \cup \text{load-tmps} \ xs) \ ys \land \text{load-tmps} \ ys \cap \bigcup (\text{fst} \ ' \ \text{store-sops} \ xs) = \{} \) \]

\textit{proof}

\textbf{locale} \ \text{valid-data-dependency} =
\textbf{fixes} \ \text{ts}:: (\ 'p, \ 'p \ \text{store-buffer}, \ \text{bool}, \text{owns}, \text{rels}) \ \text{thread-config} \ \text{list}
\textbf{assumes} \ \text{data-dependency-consistent-instrs}:
\[ \forall i \ p \ \text{is} \ \mathcal{O} \ D \not\emptyset \ \text{sb}. \]
\[ [i < \text{length} \ \text{ts}; \ \text{ts}!i = (p, \text{is}, \emptyset, \text{sb}, D, \mathcal{O}, R) ] \implies \text{data-dependency-consistent-instrs} \ (\text{dom} \ \emptyset) \ \text{is} \]
\textbf{assumes} \ \text{load-tmps-write-tmps-distinct}:
\[ \forall i \ p \ \text{is} \ \mathcal{O} \ D \not\emptyset \ \text{sb}. \]
\[ [i < \text{length} \ \text{ts}; \ \text{ts}!i = (p, \text{is}, \emptyset, \text{sb}, D, \mathcal{O}, R) ] \implies \text{load-tmps} \ \cap \bigcup (\text{fst} \ ' \ \text{write-sops} \ \text{sb}) = \{} \]

\textbf{locale} \ \text{load-tmps-fresh} =
\textbf{fixes} \ \text{ts}:: (\ 'p, \ 'p \ \text{store-buffer}, \ \text{bool}, \text{owns}, \text{rels}) \ \text{thread-config} \ \text{list}
\textbf{assumes} \ \text{load-tmps-fresh}:
\[ \forall i \ p \ \text{is} \ \mathcal{O} \ D \not\emptyset \ \text{sb}. \]
\[ [i < \text{length} \ \text{ts}; \ \text{ts}!i = (p, \text{is}, \emptyset, \text{sb}, D, \mathcal{O}, R) ] \implies \text{load-tmps} \ \cap \ \text{dom} \ \emptyset = \{} \]

\textbf{fun} \ \text{acquired-by-instrs} :: \ \text{instrs} \Rightarrow \ \text{addr set} \Rightarrow \ \text{addr set}
\textbf{where}
\[ \text{acquired-by-instrs} \ [\ A = A \]
\[ \text{acquired-by-instrs} \ (i#is) \ A = \]
\[ \text{case} \ i \ \text{of} \]
\[ \text{Read} \ - \ - \ - \Rightarrow \ \text{acquired-by-instrs} \ \text{is} \ A \]
\[ \text{Write} \ \text{volatile} \ - \ - \ \Rightarrow \ \text{acquired-by-instrs} \ \text{is} \ (\text{if} \ \text{volatile} \ \text{then} \ (A \cup A' - R) \ \text{else} \ A) \]
\[ \text{RMW} \ a \ t \ \text{sop} \ \text{cond} \ \text{ret} \ A' \ L \ R \ W \ \Rightarrow \ \text{acquired-by-instrs} \ \text{is} \ \{} \]
\[ \text{Fence} \ \Rightarrow \ \text{acquired-by-instrs} \ \text{is} \ \{} \]
\[ \text{Ghost} \ A' \ L \ R \ W \ \Rightarrow \ \text{acquired-by-instrs} \ \text{is} \ (A \cup A' - R) \]

\textbf{fun} \ \text{acquired-loads} :: \ \text{bool} \Rightarrow \ \text{instrs} \Rightarrow \ \text{addr set} \Rightarrow \ \text{addr set}
\textbf{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:
\[ \bigwedge A B. A \subseteq B \implies \text{acquired-by-instrs is } A \subseteq \text{acquired-by-instrs is } B \]
⟨proof⟩

lemma acquired-by-instrs-mono-in:
  assumes x-in: x ∈ acquired-by-instrs is A
  assumes sub: A ⊆ B
  shows x ∈ acquired-by-instrs is B
⟨proof⟩

locale enough-flushs =
  fixes ts::('p store-buffer, bool, owns, rels) thread-config list
  assumes clean-no-outstanding-volatile-Write sug:
  \[ \forall i p \in \mathcal{O} R D \not\in sb. 
    [i < \text{length ts}; tsli = (p, is, \theta, sb, D, O, R); ¬ D] \implies 
    (\text{outstanding-refs is-volatile-Write sug sb = {}}) \]

fun prog-instrs:: 'p store-buffer ⇒ instrs
where
  prog-instrs [] = []
| prog-instrs (i#is) = (case i of
  Progsug - is′ ⇒ is′ @ prog-instrs is
  | - ⇒ prog-instrs is)

fun instrs:: 'p store-buffer ⇒ instrs
where
  instrs [] = []
| instrs (i#is) = (case i of
  Write sug volatile a sop v A L R W ⇒ Write volatile a sop A L R W# instrs is
  | Read sug volatile a t v ⇒ Read volatile a t # instrs is
  | Ghost sug A L R W ⇒ Ghost A L R W# instrs is
  | - ⇒ instrs is)

locale causal-program-history =
  fixes is sug and sb
  assumes causal-program-history:
\[ \forall \text{sb}_1 \text{ sb}_2. \text{ sb}_1 = \text{ sb}_2 \implies \exists \text{ is}. \text{ instrs} \text{ sb}_2 \circ \text{ is}_{\text{sb}} = \text{ is} \circ \text{ prog-instrs} \text{ sb}_2 \]

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

**(proof)**

**Lemma** causal-program-history-suffix:
causal-program-history is sb \( \circ \) sb' \( \implies \) causal-program-history is sb sb'

**(proof)**

**Locale** valid-program-history =
fixes ts::('p, p store-buffer, bool, owns, rels) thread-config list
assumes valid-program-history:
\[ \forall i \ p \ \text{ord} \ \theta \ \text{sb}. \ (i < \text{length ts}; \ \text{ts}!i = (p, \text{is}, \theta, \text{sb}, D, O, R)) \implies \text{causal-program-history is sb} \]
assumes valid-last-prog:
\[ \forall i \ p \ \text{ord} \ \theta \ \text{sb}. \ (i < \text{length ts}; \ \text{ts}!i = (p, \text{is}, \theta, \text{sb}, D, O, R)) \implies \text{last-prog p sb = p} \]

**Lemma** (in valid-program-history) valid-program-history-nth-update:
\[ [i < \text{length ts}; \ \text{causal-program-history is sb}; \ \text{last-prog p sb = p}] \implies \text{valid-program-history (ts [i := (p, \text{is}, \theta, \text{sb}, D, O, R)])} \]

**(proof)**

**Lemma** (in outstanding-non-volatile-refs-owned-or-read-only)
outstanding-non-volatile-refs-owned-instructions-read-value-independent:
\[ \forall i \ p \ \text{ord} \ \theta \ \text{sb}. \ (i < \text{length ts}; \ \text{ts}!i = (p, \text{is}, \theta, \text{sb}, D, O, R)) \implies \text{outstanding-non-volatile-refs-owned-or-read-only S (ts[i := (p', \text{is}', \theta', \text{sb}, D', O, R'])}} \]

**(proof)**

**Lemma** (in outstanding-non-volatile-refs-owned-or-read-only)
outstanding-non-volatile-refs-owned-or-read-only-nth-update:
\[ \forall i \text{ is} \ D R \ \varnothing \ \text{sb}. \ (i < \text{length ts}; \ \text{non-volatile-owned-or-read-only False S O sb}] \implies \text{outstanding-non-volatile-refs-owned-or-read-only S (ts[i := (p, \text{is}, \theta, \text{sb}, D, O, R)])} \]

**(proof)**

**Lemma** (in outstanding-volatile-writes-unowned-by-others)
outstanding-volatile-writes-unowned-by-others-instructions-read-value-independent:
\[ \forall i \ p \ \text{ord} \ \theta \ \text{sb}. \ (i < \text{length ts}; \ \text{ts}!i = (p, \text{is}, \theta, \text{sb}, D, O, R)) \implies \text{outstanding-volatile-writes-unowned-by-others (ts[i := (p', \text{is}', \theta', \text{sb}, D', O, R'])}} \]

**(proof)**

**Lemma** (in read-only-reads-unowned)
read-only-unowned-instructions-read-value-independent:

79
\begin{align*}
\forall i \ p & \text{ is } O \ R \ D \ \varnothing \ \text{sb.}
\forall i < \text{length ts; ts}!i = (p,\text{is},\varnothing,\text{sb},D,O,R) \ [\implies \\
& \text{read-only-reads-unowned ts[i} := (p',\text{is}',\varnothing',\text{sb},D',O,R')).]
& \langle \text{proof} \rangle
\end{align*}

\textbf{lemma Write}_{sb}\text{-in-outstanding-refs:}

Write}_{sb}\text{True} \text{ a sop } V \ A \ L \ R \ W \in \text{set xs} \implies \text{a } \in \text{outstanding-refs is-volatile-Write}_{sb}\text{ xs}

& \langle \text{proof} \rangle
\end{align*}

\textbf{lemma (in outstanding-volatile-writes-unowned-by-others)}

outstanding-volatile-writes-unowned-by-others-store-buffer:

\begin{align*}
\forall i \ p & \text{ is } O \ R \ D \ \varnothing \ \text{sb.}
\forall i < \text{length ts; ts}!i = (p,\text{is},\varnothing,\text{sb},D,O,R); \\
& \text{all-acquired sb}' \subseteq \text{all-acquired sb]} \implies \\
& \text{outstanding-volatile-writes-unowned-by-others ts[i} := (p',\text{is}',\varnothing',\text{sb},D',O,R')).]
& \langle \text{proof} \rangle
\end{align*}

\textbf{lemma (in ownership-distinct)}

ownership-distinct-instructions-read-value-store-buffer-independent:

\begin{align*}
\forall i \ p & \text{ is } O \ R \ D \ \varnothing \ \text{sb.}
\forall i < \text{length ts; ts}!i = (p,\text{is},\varnothing,\text{sb},D,O,R); \\
& \text{all-acquired sb}' \subseteq \text{all-acquired sb]} \implies \\
& \text{ownership-distinct ts[i} := (p',\text{is}',\varnothing',\text{sb},D',O,R')).]
& \langle \text{proof} \rangle
\end{align*}

\textbf{lemma (in ownership-distinct)}

ownership-distinct-nth-update:

\begin{align*}
\forall i \ p & \text{ is } O \ R \ D \ \varnothing \ \text{sb.}
\forall i < \text{length ts; ts}!i = (p,\text{is},\varnothing,\text{sb},D,O,R); \\
& \forall j < \text{length ts. } i \neq j \longrightarrow (\text{let (p}_j,\text{is}_j,\varnothing,\text{sb}_j,D_j,O_j,R_j) = ts!j \\
& \text{in (O' } \cup \text{ all-acquired sb}') \cap (O_j \cup \text{all-acquired sb}_j) = \{} )] \implies \\
& \text{ownership-distinct ts[i} := (p',\text{is}',\varnothing',\text{sb},D',O',R')).]
& \langle \text{proof} \rangle
\end{align*}

\textbf{lemma (in valid-write-sops)}

valid-write-sops-nth-update:

\begin{align*}
\forall i < \text{length ts; } \forall \text{sop } \in \text{write-sops sb. valid-sop sop] } \implies \\
& \text{valid-write-sops ts[i} := (p,\text{is},\text{xs},\text{sb},D,O,R)].]
& \langle \text{proof} \rangle
\end{align*}

\textbf{lemma (in valid-store-sops)}

valid-store-sops-nth-update:

\begin{align*}
\forall i < \text{length ts; } \forall \text{sop } \in \text{store-sops is. valid-sop sop] } \implies \\
& \text{valid-store-sops ts[i} := (p,\text{is},\text{xs},\text{sb},D,O,R)].]
& \langle \text{proof} \rangle
\end{align*}
**Lemma (in valid-sops) valid-sops-nth-update:**

\[ [i < \text{length } ts; \forall \text{sop } \in \text{write-sops } sb. \text{valid-sop } \text{sop}; \]
\[ \forall \text{sop } \in \text{store-sops } is. \text{valid-sop } \text{sop}] \implies\]
\[ \text{valid-sops } (ts[i := (p,is,xs,sb,D,O,R)])] \]

**(proof)**

**Lemma (in valid-data-dependency) valid-data-dependency-nth-update:**

\[ [i < \text{length } ts; \text{data-dependency-consistent-instrs } (\text{dom } \emptyset) \text{ is;}
\]
\[ \text{load-tmps } is \cap \bigcup (\text{fst } \text{write-sops } sb) = \emptyset]\]
\[ \implies \text{valid-data-dependency } (ts[i := (p,is,\emptyset,sb,D,O,R)])] \]

**(proof)**

**Lemma (in enough-flushs) enough-flushs-nth-update:**

\[ [i < \text{length } ts;
\]
\[ \neg D \rightarrow (\text{outstanding-refs is-volatile-Write}_{sb} \text{ sb } = \emptyset)\]
\[ \implies \text{enough-flushs } (ts[i := (p,is,\emptyset,sb,D,O,R)])] \]

**(proof)**

**Lemma (in outstanding-non-volatile-writes-unshared) outstanding-non-volatile-writes-unshared-nth-update:**

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

**(proof)**

**Lemma (in sharing-consis) sharing-consis-nth-update:**

\[ [i < \text{length } ts; \text{sharing-consistent } S \text{ O sb}] \implies\]
\[ \text{sharing-consis } S (ts[i := (p,is,xs,sb,D,O,R)])] \]

**(proof)**

**Lemma (in no-outstanding-write-to-read-only-memory) no-outstanding-write-to-read-only-memory-nth-update:**

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

**(proof)**

**Lemma** in-Union-image-nth-conv: \( a \in \bigcup (f \uparrow \text{set } xs) \implies \exists i. i < \text{length } xs \land a \in f (xs![i]) \)

**(proof)**

**Lemma** in-Inter-image-nth-conv: \( a \in \bigcap (f \uparrow \text{set } xs) = (\forall i < \text{length } xs. a \in f (xs![i])) \)

**(proof)**
lemma release-ownership-nth-update:
assumes R-subset: R ⊆ O
shows \( \land i. \ [i < \text{length } ts; \ ts[i] = (p, is, xs, sb, D, O, R)\]
    \land \text{ownership-distinct } ts\]
    \[\implies \bigcup ((\lambda(-,-,-,O,r). O) \setminus \text{set } ts[i := (p', is', xs', sb', D', O \cup A, R')])\]
    \[\text{=} \left((\bigcup ((\lambda(-,-,-,O,r). O) \setminus \text{set } ts)) \cup A\right)\]
\langle proof \rangle

lemma acquire-ownership-nth-update:
shows \( \land i. \ [i < \text{length } ts; \ ts[i] = (p, is, xs, sb, D, O, R)\]
    \land \text{ownership-distinct } ts\]
    \[\implies \bigcup ((\lambda(-,-,-,O,r). O) \setminus \text{set } ts[i := (p', is', xs', sb', D', O \cup A, R')])\]
    \[\text{=} \left((\bigcup ((\lambda(-,-,-,O,r). O) \setminus \text{set } ts)) \cup A\right)\]
\langle proof \rangle

lemma acquire-release-ownership-nth-update:
assumes R-subset: R ⊆ O
shows \( \land i. \ [i < \text{length } ts; \ ts[i] = (p, is, xs, sb, D, O, R)\]
    \land \text{ownership-distinct } ts\]
    \[\implies \bigcup ((\lambda(-,-,-,O,r). O) \setminus \text{set } ts[i := (p', is', xs', sb', D', O \cup A, R')])\]
    \[\text{=} \left((\bigcup ((\lambda(-,-,-,O,r). O) \setminus \text{set } ts)) \cup A\right)\]
\langle proof \rangle

lemma (in valid-history) valid-history-nth-update:
\[ \ [i < \text{length } ts; \ \text{history-consistent } \emptyset \ (\text{hd-prog } p \ \text{sb}) \ \text{sb} \] \[\implies \text{valid-history program-step } (ts[i := (p, is, \emptyset, sb, D, O, R)])\]
\langle proof \rangle

lemma (in valid-reads) valid-reads-nth-update:
\[ \ [i < \text{length } ts; \ \text{reads-consistent } False \ O \ m \ \text{sb} \] \[\implies \text{valid-reads } m \ (ts[i := (p, is, xs, sb, D, O, R)])\]
\langle proof \rangle

lemma (in load-tmps-distinct) load-tmps-distinct-nth-update:
\[ \ [i < \text{length } ts; \ \text{distinct-load-tmps } is\] \[\implies \text{load-tmps-distinct } (ts[i := (p, is, xs, sb, D, O, R)])\]
\langle proof \rangle

lemma (in read-tmps-distinct) read-tmps-distinct-nth-update:
\[ \ [i < \text{length } ts; \ \text{distinct-read-tmps } sb\] \[\implies \text{read-tmps-distinct } (ts[i := (p, is, xs, sb, D, O, R)])\]
\langle proof \rangle

lemma (in load-tmps-read-tmps-distinct) load-tmps-read-tmps-distinct-nth-update:
\[ \ [i < \text{length } ts; \ \text{load-tmps } is \cap \text{read-tmps } sb = \{\} \] \[\implies \text{load-tmps-read-tmps-distinct } (ts[i := (p, is, xs, sb, D, O, R)])\]
\langle proof \rangle

lemma (in load-tmps-fresh) load-tmps-fresh-nth-update:
\[ i < \text{length} \, \text{ts}; \]
\[ \text{load-tmps} \, \cap \, \text{dom} \, \emptyset = \{\} \implies \]
\[ \text{load-tmps-fresh} \ (\text{ts}[i := (p, is, \emptyset, sb, D, O, R)]) \]

\textbf{fun} flush-all-until-volatile-write::
\[
(\ 'p, 'p \text{ store-buffer}, 'd \text{ irty}, 'o \text{ wns}, 'r \text{ els}) \ \text{thread-config list} \implies \text{memory} \implies \text{memory}
\]
\textbf{where}
\[
\begin{align*}
\text{flush-all-until-volatile-write} \ [ \ m &= m \\
\text{flush-all-until-volatile-write} \ ((\cdot, \cdot, \cdot, \cdot, sb, \cdot, \cdot) \# \text{ts}) \ m &= \\
& \text{flush-all-until-volatile-write} \ \text{ts} \ (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ m)
\end{align*}
\]

\textbf{fun} share-all-until-volatile-write::
\[
(\ 'p, 'p \text{ store-buffer}, 'd \text{ irty}, 'o \text{ wns}, 'r \text{ els}) \ \text{thread-config list} \implies \text{shared} \implies \text{shared}
\]
\textbf{where}
\[
\begin{align*}
\text{share-all-until-volatile-write} \ [ \ S &= S \\
\text{share-all-until-volatile-write} \ ((\cdot, \cdot, \cdot, \cdot, sb, \cdot, \cdot) \# \text{ts}) \ S &= \\
& \text{share-all-until-volatile-write} \ \text{ts} \ (\text{share} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ S)
\end{align*}
\]

\textbf{lemma} takeWhile-dropWhile-real-prefix:
\[
[x \in \text{set} \ \text{xs}; \ \neg \ P \ x] \implies \exists \ y \ \text{ys}. \ \text{xs} = \text{takeWhile} \ P \ \text{xs} @ y\#\text{ys} \land \neg \ P \ y \land \text{dropWhile} \ P \ \text{xs}
\]
\textbf{proof}

\textbf{lemma} buffered-val-witness: buffered-val \ sb \ a = \text{Some} \ v \implies \exists \ \text{volatile} \ sop \ A \ L \ R \ W. \ \text{Write}_{sb} \ \text{volatile} \ a \ \text{v} \ A \ L \ R \ W \in \text{set} \ sb
\textbf{proof}

\textbf{lemma} flush-append-Read_{sb}:
\[
\forall \ m. \ (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (\text{sb} @ [\text{Read}_{sb} \ \text{volatile} \ a \ t \ v]))) \ m)
\]
\[
= \text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ m
\]
\textbf{proof}

\textbf{lemma} flush-append-write:
\[
\forall \ m. \ (\text{flush} \ (\text{sb} @ [\text{Write}_{sb} \ \text{volatile} \ a \ \text{v} \ A \ L \ R \ W])) \ m) = (\text{flush} \ \text{sb} \ m) \ (a:=v)
\]
\textbf{proof}

\textbf{lemma} flush-append-Prog_{sb}:
\[
\forall \ m. \ (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ (\text{sb} @ [\text{Prog}_{sb} \ p_1 \ p_2 \ \text{mis}]))) \ m)
\]
\[
= \ (\text{flush} \ (\text{takeWhile} \ (\text{Not} \circ \text{is-volatile-Write}_{sb}) \ \text{sb}) \ m)
\]
\textbf{proof}
**Lemma** flush-append-Ghost$_{ab}$:
\[ \forall m. \text{flush} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) (sb \sqcup \text{Ghost}_b \text{A L R W}]) m) = \text{flush} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) sb) m \]

\((\text{proof})\)

**Lemma** share-append:
\[ \forall S. \text{share} (xs @ ys) S = \text{share} ys (\text{share} xs S) \]

\((\text{proof})\)

**Lemma** share-append-Read$_{ab}$:
\[ \forall S. (\text{share} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) (sb \sqcup \text{Read}_b \text{volatile a t v})) S) = \text{share} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) sb) S \]

\((\text{proof})\)

**Lemma** share-append-Write$_{ab}$:
\[ \forall S. (\text{share} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) (sb \sqcup \text{Write}_b \text{volatile a sop v A L R W})) S) = \text{share} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) sb) S \]

\((\text{proof})\)

**Lemma** share-append-Prog$_{ab}$:
\[ \forall S. (\text{share} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) (sb \sqcup \text{Prog}_b \text{p1 p2 mis})) S) = (\text{share} (\text{takeWhile} (\neg \circ \text{is-volatile-Write}_b) sb) S) \]

\((\text{proof})\)

**Lemma** in-acquired-no-pending-write-outstanding-write:
\[ a \in \text{acquired False sb A} \implies \text{outstanding-refs is-volatile-Write}_b \text{sb} \neq \{\} \]

\((\text{proof})\)

**Lemma** flush-buffered-val-conv:
\[ \forall m. \text{flush} sb m a = (\text{case} \text{buffered-val} sb a \text{of None} \implies m a \mid \text{Some} v \implies v) \]

\((\text{proof})\)

**Lemma** reads-consistent-unbuffered-snoc:
\[ \forall m. \text{buffered-val} sb a = \text{None} \implies m a = v \implies \text{reads-consistent pending-write O m sb} \implies \text{volatile} \implies \text{outstanding-refs is-volatile-Write}_b \text{sb} = \{\} \implies \text{reads-consistent pending-write O m sb} \]

\((\text{proof})\)

**Lemma** reads-consistent-buffered-snoc:
\[ \forall m. \text{buffered-val} sb a = \text{Some} v \implies \text{reads-consistent pending-write O m sb} \implies \text{volatile} \implies \text{outstanding-refs is-volatile-Write}_b \text{sb} = \{\} \implies \text{reads-consistent pending-write O m sb} \]

\((\text{proof})\)

**Lemma** reads-consistent-snoc-Write$_{ab}$:
\[ \forall m. \text{reads-consistent pending-write O m sb} \implies \]
proof

lemma reads-consistent-snoc-Prog\sb:
\[ \forall m. \text{reads-consistent pending-write } O \ m \ \text{sb} \implies \text{reads-consistent pending-write } O \ m \ (\text{sb} \ \oplus \ \text{Prog}_{sb} \ p_1 \ p_2 \ \text{mis}) \]
(proof)

lemma reads-consistent-snoc-Ghost\sb:
\[ \forall m. \text{reads-consistent pending-write } O \ m \ \text{sb} \implies \text{reads-consistent pending-write } O \ m \ (\text{sb} \ \oplus \ \text{Ghost}_{sb} \ \text{A L R W}) \]
(proof)

lemma restrict-map-id [simp]:\[ m \restriction \text{dom } m = m \]
(proof)

lemma flush-all-until-volatile-write-Read-commute:
shows \[ \forall m. i < \text{length } ls; \ ls[i]=(p, \text{Read volatile a t#is}, \emptyset, sb, D, O, R) \]
\[ \implies \text{flush-all-until-volatile-write} \]
\[ (ls[i] := (p, \text{is}, \emptyset(t\mapsto v), sb \ \oplus \ \text{Read}_{sb} \ \text{volatile a t v}, D', O', R')) \ m = \]
\[ \text{flush-all-until-volatile-write} \ ] \]
ls m
(proof)

lemma flush-all-until-volatile-write-append-Ghost-commute:
\[ \forall i \ m. \ [i < \text{length } ts; ts[i]=(p, \text{is}, \emptyset, sb, D, O, R)] \]
\[ \implies \text{flush-all-until-volatile-write} \]
\[ (ts[i] := (p', \text{is}', \emptyset', sb \ \oplus \ \text{Ghost}_{sb} \ \text{A L R W}, D', O', R')) \ ] \]
m
\[ = \text{flush-all-until-volatile-write} \]
\[ ts \ m \]
(proof)

lemma update-commute:
assumes g-unchanged: \[ \forall a \ m. \ a \notin G \implies g \ m \ a = m \ a \]
assumes g-independent: \[ \forall a \ m. \ a \in G \implies g \ (f \ m) \ a = g \ m \ a \]
assumes f-unchanged: \[ \forall a \ m. \ a \notin F \implies f \ m \ a = m \ a \]
assumes f-independent: \[ \forall a \ m. \ a \in F \implies f \ (g \ m) \ a = f \ m \ a \]
assumes disj: \[ G \cap F = \{\} \]
shows \[ f \ (g \ m) = g \ (f \ m) \]
(proof)

lemma update-commute’:
assumes g-unchanged: \[ \forall a \ m. \ a \notin G \implies g \ m \ a = m \ a \]
assumes g-independent: \[ \forall a \ m_1 \ m_2. \ a \in G \implies g \ m_1 \ a = g \ m_2 \ a \]
assumes f-unchanged: \[ \forall a \ m. \ a \notin F \implies f \ m \ a = m \ a \]
assumes f-independent: \[ \forall a \ m_1 \ m_2. \ a \in F \implies f \ m_1 \ a = f \ m_2 \ a \]
assumes disj: \[ G \cap F = \{\} \]
\[ f(g(m)) = g(f(m)) \]

**lemma** flush-unchanged-addresses: \( \forall m. a \notin \text{outstanding-refs is-Write}_{sb} \Rightarrow \text{flush}_{sb} m a = m a \)

**lemma** flushed-values-mem-independent:
\[ \forall m m'. a \in \text{outstanding-refs is-Write}_{sb} \Rightarrow \text{flush}_{sb} m a = \text{flush}_{sb} m' a \]

**lemma** flush-all-until-volatile-write-unchanged-addresses:
\[ \forall m. a \notin \bigcup((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{outstanding-refs is-Write}_{sb})
\quad (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \ ') \text{ set ls}) \Rightarrow \\
\quad \text{flush-all-until-volatile-write} \text{ is } m a = m a \]

**lemma** notin-outstanding-non-volatile-takeWhile-lem:
\[ a \notin \text{outstanding-refs (Not} \circ \text{is-volatile}) sb \Rightarrow \]
\[ a \notin \text{outstanding-refs is-Write}_{sb} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \]

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

**lemma** notin-outstanding-non-volatile-takeWhile-Un-lem:
\[ a \notin \bigcup((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{outstanding-refs is-volatile-Write}_{sb}) \ ') \text{ set ls}) \Rightarrow \\
\quad a \notin \bigcup((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{outstanding-refs is-Write}_{sb})
\quad (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_{sb}) sb) \ ') \text{ set ls}) \]

**lemma** flush-all-until-volatile-write-unchanged-addresses':
\[ \text{assumes no-volatile-Write}_{sb}: \text{outstanding-refs is-volatile-Write}_{sb} sb = \{\} \]
\[ \text{shows } \forall m. i < \text{length ls}; \text{ls}! i = (p, is, xs, sb, D, O, R); \]
∀ \( j < \text{length } \text{ls} \). i \neq j \rightarrow \\
(\text{let } (-,\ldots,\text{sb}_j,\ldots) = \text{ls}[j] \\
\quad \text{in } a \notin \text{outstanding-refs} \text{ is-non-volatile-Write}_{\text{sb}} \text{ (takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}_j) \text{ )} \rightarrow \\
\quad \text{flush-all-until-volatile-write } \text{ls } m \ a = \\
\quad (\text{case } \text{buffered-val } \text{sb} a \text{ of } \text{None } \Rightarrow m \ a \mid \text{Some } v \Rightarrow v) \\
\langle \text{proof} \rangle

\text{context program}
\text{begin}

\text{abbreviation } \text{sb-concurrent-step} :: \\
\quad (\text{'p},\text{'p store-buffer,dirty,owns,rels,shared}) \text{ global-config } \Rightarrow (\text{'p},\text{'p store-buffer,dirty,owns,rels,shared}) \text{ global-config } \Rightarrow \text{bool} \\
\quad (\text{\sim } \Rightarrow_{\text{sb}} - [60,60] 100)
\text{where}
\quad \text{sb-concurrent-step } \equiv \\
\quad \text{computation.concurrent-step } \text{sb-memop-step store-buffer-step program-step } (\lambda \text{p p}' \text{ is sb. sb})

\text{term} \ x \Rightarrow_{\text{sb}} Y

\text{abbreviation} \ (\text{in program}) \text{ sb-concurrent-steps} :: \\
\quad (\text{'p},\text{'p store-buffer,dirty,owns,rels,shared}) \text{ global-config } \Rightarrow (\text{'p},\text{'p store-buffer,dirty,owns,rels,shared}) \text{ global-config } \Rightarrow \text{bool} \\
\quad (\text{\sim } \Rightarrow_{\text{sb}^*} - [60,60] 100)
\text{where}
\quad \text{sb-concurrent-steps } \equiv \text{sb-concurrent-step}^*^{**}

\text{term} \ x \Rightarrow_{\text{sb}^*} Y

\text{abbreviation } \text{sbh-concurrent-step} :: \\
\quad (\text{'p},\text{'p store-buffer,bool,owns,rels,shared}) \text{ global-config } \Rightarrow (\text{'p},\text{'p store-buffer,bool,owns,rels,shared}) \text{ global-config } \Rightarrow \text{bool} \\
\quad (\text{\sim } \Rightarrow_{\text{sbh}} - [60,60] 100)
\text{where}
\quad \text{sbh-concurrent-step } \equiv \\
\quad \text{computation.concurrent-step } \text{sbh-memop-step flush-step program-step } (\lambda \text{p p}' \text{ is sb. sb @ [Prog}_{\text{sb}} \text{ p p}' \text{ is] } )

\text{term} \ x \Rightarrow_{\text{sbh}} Y

\text{abbreviation} \ \text{sbh-concurrent-steps} :: \\
\quad (\text{'p},\text{'p store-buffer,bool,owns,rels,shared}) \text{ global-config } \Rightarrow (\text{'p},\text{'p store-buffer,bool,owns,rels,shared}) \text{ global-config } \Rightarrow \text{bool} \\
\quad (\text{\sim } \Rightarrow_{\text{sbh}^*} - [60,60] 100)
\text{where}
\quad \text{sbh-concurrent-steps } \equiv \text{sbh-concurrent-step}^*^{**}

87
term \( x \Rightarrow_{sb}^* Y \)

end

lemma instrs-append-Read_{sb}:
\[ \text{instrs} (sb@[\text{Read}_{sb} \text{ volatile a t v}]) = \text{instrs} \; sb \; @ \; [\text{Read} \text{ volatile a t}] \]
(\text{proof})

lemma instrs-append-Write_{sb}:
\[ \text{instrs} (sb@[\text{Write}_{sb} \text{ volatile a sop v A L R W}]) = \text{instrs} \; sb \; @ \; [\text{Write} \text{ volatile a sop A L R W}] \]
(\text{proof})

lemma instrs-append-Ghost_{sb}:
\[ \text{instrs} (sb@[\text{Ghost}_{sb} \text{ A L R W}]) = \text{instrs} \; sb \; @ \; [\text{Ghost} \text{ A L R W}] \]
(\text{proof})

lemma prog-instrs-append-Ghost_{sb}:
\[ \text{prog-instrs} (sb@[\text{Ghost}_{sb} \text{ A L R W}]) = \text{prog-instrs} \; sb \]
(\text{proof})

lemma prog-instrs-append-Read_{sb}:
\[ \text{prog-instrs} (sb@[\text{Read}_{sb} \text{ volatile a t v}]) = \text{prog-instrs} \; sb \]
(\text{proof})

lemma prog-instrs-append-Write_{sb}:
\[ \text{prog-instrs} (sb@[\text{Write}_{sb} \text{ volatile a sop v A L R W}]) = \text{prog-instrs} \; sb \]
(\text{proof})

lemma hd-prog-append-Read_{sb}:
\[ \text{hd-prog} \; p \; (sb@[\text{Read}_{sb} \text{ volatile a t v}]) = \text{hd-prog} \; p \; sb \]
(\text{proof})

lemma hd-prog-append-Write_{sb}:
\[ \text{hd-prog} \; p \; (sb@[\text{Write}_{sb} \text{ volatile a sop v A L R W}]) = \text{hd-prog} \; p \; sb \]
(\text{proof})

lemma flush-update-other: \( \forall m. a \notin \text{outstanding-refs} \; (\text{Not} \; \circ \; \text{is-volatile}) \; sb \implies \text{outstanding-refs} \; (\text{is-volatile-Write}_{sb}) \; sb = \{ \} \implies \text{flush} \; sb \; (m(a:=v)) = (\text{flush} \; sb \; m)(a := v) \)
(\text{proof})

lemma flush-update-other: \( \forall m. a \notin \text{outstanding-refs} \; (\text{is-non-volatile-Write}_{sb}) \; sb \implies \text{outstanding-refs} \; (\text{is-volatile-Write}_{sb}) \; sb = \{ \} \implies \text{flush} \; sb \; (m(a:=v)) = (\text{flush} \; sb \; m)(a := v) \)
(\text{proof})

lemma flush-update-other: \( \forall m. a \notin \text{outstanding-refs} \; (\text{is-non-volatile-Write}_{sb}) \; sb \implies a \notin \text{outstanding-refs} \; (\text{is-volatile-Write}_{sb}) \; sb \implies \text{flush} \; sb \; (m(a:=v)) = (\text{flush} \; sb \; m)(a := v) \)
lemma flush-all-until-volatile-write-update-other:
\[ \forall m. \forall j < \text{length} \ ts. \]
\[ (\text{let } (-,-,-,sb_j,-,-) = ts!j \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb_j} \text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb_j}) sb))} \]
\[ \implies \text{flush-all-until-volatile-write} ts (m(a := v)) = \]
\[ (\text{flush-all-until-volatile-write} \ ts m)(a := v) \]
\[ \langle proof \rangle \]

lemma flush-all-until-volatile-write-append-non-volatile-write-commute:
assumes no-volatile-Write_{sb}: \text{outstanding-refs is-volatile-Write}_{sb} sb = {} 
shows \[ \forall m i. \ [i < \text{length} \ ts; ts[i] = (p,is,\theta,\theta,\theta) ; \]
\[ \forall j < \text{length} \ ts. \ i \neq j \implies \]
\[ (\text{let } (-,-,-,sb_j,-,-) = ts!j \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb_j} \text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb_j}) sb))} \]
\[ \implies \text{flush-all-until-volatile-write} (ts[i := (p',is', \theta, sb @ \{\text{Write}_{sb} \theta a sop v A L R W\},D', \theta,\theta,\theta,\theta'])] m = \]
\[ (\text{flush-all-until-volatile-write} \ ts m)(a := v) \]
\[ \langle proof \rangle \]

lemma flush-all-until-volatile-write-append-unflushed:
assumes volatile-Write_{sb}: \neg \text{outstanding-refs is-volatile-Write}_{sb} sb = {} 
shows \[ \forall m i. \ [i < \text{length} \ ts; ts[i] = (p,is,\theta,\theta,\theta) ; \]
\[ \forall j < \text{length} \ ts. \ i \neq j \implies \]
\[ (\text{let } (-,-,-,sb_j,-,-) = ts!j \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb_j} \text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb_j}) sb))} \]
\[ \implies \text{flush-all-until-volatile-write} \ (ts[i := (p',is', \theta, sb @ sbx,D', \theta,\theta,\theta,\theta'])] m = \]
\[ (\text{flush-all-until-volatile-write} \ ts m) \]
\[ \langle proof \rangle \]

lemma flush-all-until-volatile-nth-update-unused:
shows \[ \forall m i. \ [i < \text{length} \ ts; ts[i] = (p,is,\theta,\theta,\theta) ; \]
\[ \forall j < \text{length} \ ts. \ i \neq j \implies \]
\[ (\text{let } (-,-,-,sb_j,-,-) = ts!j \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb_j} \text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb_j}) sb))} \]
\[ \implies \text{flush-all-until-volatile-write} \ (ts[i := (p',is', \theta, sb @ \{\text{Write}_{sb} \theta a sop v A L R W\},D', \theta,\theta,\theta,\theta'])] m = \]
\[ (\text{flush-all-until-volatile-write} \ ts m) \]
\[ \langle proof \rangle \]

lemma flush-all-until-volatile-write-append-volatile-write-commute:
assumes no-volatile-Write_{sb}: \text{outstanding-refs is-volatile-Write}_{sb} sb = {} 
shows \[ \forall m i. \ [i < \text{length} \ ts; ts[i] = (p,is,\theta,\theta,\theta) ; \]
\[ \forall j < \text{length} \ ts. \ i \neq j \implies \]
\[ (\text{let } (-,-,-,sb_j,-,-) = ts!j \in a \notin \text{outstanding-refs is-non-volatile-Write}_{sb_j} \text{ (takeWhile (Not \circ \text{is-volatile-Write}_{sb_j}) sb))} \]
\[ \implies \text{flush-all-until-volatile-write} \ (ts[i := (p',is', \theta, sb @ \{\text{Write}_{sb} \theta a sop v A L R W\},D', \theta,\theta,\theta,\theta'])] m = \]
\[ \text{flush-all-until-volatile-write} \ ts m \]
\[ \langle proof \rangle \]

lemma reads-consistent-update:
\[ \forall \text{pending-write } O m. \ \text{reads-consistent pending-write } O m \text{ sb} \implies \]
\[ a \notin \text{outstanding-refs (Not \circ \text{is-volatile}) sb} \implies \]
\[ \text{reads-consistent pending-write } O (m(a := v)) \text{ sb} \]
\[ \langle proof \rangle \]
**lemma (in program)** history-consistent-hd-prog:
\[ \forall p. \text{history-consistent } \vartheta p' \text{ xs } \implies \text{history-consistent } \vartheta (\text{hd-prog } p \text{ xs}) \text{ xs} \]

\langle proof \rangle

**locale** valid-program = program +

**fixes** valid-prog

**assumes** valid-prog-inv: \[ \vartheta \models p \to (p', \text{is'}); \text{valid-prog } p \] \implies \text{valid-prog } p' \]

**lemma (in valid-program)** history-consistent-appendD:
\[ \forall \vartheta \text{ ys } p. \forall \text{sop } \in \text{write-sops } \text{xs}. \text{valid-sop sop } \implies \text{read-tmps } \text{xs } \cap \text{read-tmps } \text{ys } = \{ \} \implies \text{history-consistent } \vartheta \text{ p } (\text{xs@ys}) \implies \]

\[ \text{(history-consistent } (\vartheta|' (\text{dom } \vartheta - \text{read-tmps } \text{ys}))) \text{ p } \text{xs } \land \]

\[ \text{history-consistent } \vartheta (\text{last-prog } p \text{ xs}) \text{ ys } \land \]

\[ \text{read-tmps } \text{ys } \cap \bigcup (\text{fst }' \text{ write-sops } \text{xs}) = \{ \} \]

\langle proof \rangle

**lemma (in valid-program)** history-consistent-appendI:
\[ \forall \vartheta \text{ ys } p. \forall \text{sop } \in \text{write-sops } \text{xs}. \text{valid-sop sop } \implies \text{read-tmps } \text{xs } \cap \bigcup (\text{fst }' \text{ write-sops } \text{xs}) = \{ \} \implies \text{valid-prog } p \implies \]

\[ \text{history-consistent } \vartheta \text{ p } (\text{xs@ys}) \]

\langle proof \rangle

**lemma (in valid-program)** history-consistent-append-conv:
\[ \forall \vartheta \text{ ys } p. \forall \text{sop } \in \text{write-sops } \text{xs}. \text{valid-sop sop } \implies \text{read-tmps } \text{xs } \cap \bigcup (\text{fst }' \text{ write-sops } \text{xs}) = \{ \} \implies \text{valid-prog } p \implies \]

\[ \text{(history-consistent } (\vartheta|' (\text{dom } \vartheta - \text{read-tmps } \text{ys}))) \text{ p } \text{xs } \land \]

\[ \text{history-consistent } \vartheta (\text{last-prog } p \text{ xs}) \text{ ys } \land \]

\[ \text{read-tmps } \text{ys } \cap \bigcup (\text{fst }' \text{ write-sops } \text{xs}) = \{ \} \]

\langle proof \rangle

**lemma** instrs-takeWhile-dropWhile-conv:
\[
\text{instrs } \text{xs } = \text{instrs } (\text{takeWhile } P \text{ xs}) \odot \text{instrs } (\text{dropWhile } P \text{ xs})
\]

\langle proof \rangle

**lemma (in program)** history-consistent-hd-prog-p:
\[ \forall p. \text{history-consistent } \vartheta p \text{ xs } \implies p = \text{hd-prog } p \text{ xs} \]

\langle proof \rangle

**lemma** instrs-append:
\[ \forall y s. \text{instrs } (y s@y s) = \text{instrs } y s @ \text{instrs } y s \]

\langle proof \rangle

**lemma** prog-instrs-append:
\[ \forall y s. \text{prog-instrs } (y s@y s) = \text{prog-instrs } y s @ \text{prog-instrs } y s \]

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

⟨proof⟩

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

⟨proof⟩

lemma prog-instrs-filter-is-Prog_{sb}: \( \text{prog-instrs } (\text{filter } (\text{is-Prog}_{sb}) xs) = \text{prog-instrs } xs \)

⟨proof⟩

lemma Cons-to-snoc: \( \forall x. \exists ys y. (x#xs) = (ys@y) \)

⟨proof⟩

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⟩

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⟩

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} p1 p2 mis])

⟨proof⟩

lemma causal-program-history-Ghost:

assumes causal-Ghost_{sb}: causal-program-history (Ghost A L R W # is_{sb}) sb

shows causal-program-history is_{sb} (sb @ [Ghost_{sb} A L R W])

⟨proof⟩

lemma hd-prog-last-prog-end: \( [p = \text{hd-prog } p \ sb ; \text{last-prog } p \ sb = p_{sb}] \implies p = \text{hd-prog } p_{sb} \ sb \)

⟨proof⟩

lemma hd-prog-idem: \( \text{hd-prog } (\text{hd-prog } p \ xs) \ xs = \text{hd-prog } p \ xs \)

⟨proof⟩

lemma last-prog-idem: \( \text{last-prog } (\text{last-prog } p \ sb) \ sb = \text{last-prog } p \ sb \)

⟨proof⟩

lemma last-prog-hd-prog-append:

last-prog (hd-prog p_{sb} (sb@sb')) sb = last-prog (hd-prog p_{sb} sb') sb

⟨proof⟩
lemma last-prog-hd-prog: last-prog (hd-prog p xs) xs = last-prog p xs
(proof)

lemma last-prog-append-Readsb:
\[ p. \text{last-prog } p \ (sb @ [\text{Read}_{sb} \text{ volatile a t v}]) = \text{last-prog } p \ sb \]
(proof)

lemma last-prog-append-Writesb:
\[ p. \text{last-prog } p \ (sb @ [\text{Write}_{sb} \text{ volatile a sop v A L R W}]) = \text{last-prog } p \ sb \]
(proof)

lemma last-prog-append-Progsb:
\[ x. \text{last-prog } x \ (sb@[\text{Prog}_{sb} \ p \ p' \text{ mis}]) = p' \]
(proof)

lemma hd-prog-append-Progsb: hd-prog x (sb @ [\text{Prog}_{sb} \ p \ p' \text{ mis}]) = hd-prog p sb
(proof)

lemma hd-prog-last-prog-append-Progsb:
\[ p'. \text{hd-prog } p' \ x \ s = p' \implies \text{last-prog } p' \ x \ s = p_1 \implies \]
\[ \text{hd-prog } p' \ (x @ [\text{Prog}\_{sb} \ p_1 \ p_2 \text{ mis}]) = p' \]
(proof)

lemma hd-prog-append-Ghostsb:
\[ \text{hd-prog } p \ (sb@[\text{Ghost}_{sb} \ A \ R L W]) = \text{hd-prog } p \ sb \]
(proof)

lemma last-prog-append-Ghostsb:
\[ p. \text{last-prog } p \ (sb@[\text{Ghost}_{sb} \ A \ L R W]) = \text{last-prog } p \ sb \]
(proof)

lemma dropWhile-all-False-conv:
\[ \forall x \in \text{set } x. \neg P \ x \implies \text{dropWhile } P \ x \ s = x \]
(proof)

lemma dropWhile-append-all-False:
\[ \forall y \in \text{set } y. \neg P \ y \implies \]
\[ \text{dropWhile } P \ (x @ y) = \text{dropWhile } P \ x @ y \]
(proof)

lemma reads-consistent-append-first:
\[ \forall m \ y. \text{reads-consistent pending-write } \mathcal{O} \ m \ (x @ y) \implies \text{reads-consistent pending-write } \mathcal{O} \ m \ x \]
(proof)
Lemma reads-consistent-takeWhile:
assumes consis: reads-consistent pending-write $O$ m sb
shows reads-consistent pending-write $O$ (takeWhile P sb)
\[\text{proof}\]

Lemma flush-flush-all-until-volatile-write-Write\_sb-volatile-commute:
\[\forall i. \left[ i < \text{length ts} \implies ts!i = (p, is, xs, Write\_sb True a sop v A L R W #sb, D, O, R) \right] ;
\forall i < \text{length ts}. \forall j < \text{length ts}. i \neq j \implies
\begin{align*}
& \text{let } (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot) = ts!i; \\
& (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot) = ts!j \\
& \text{in outstanding-refs is-Write\_sb sb}_i \cap \\
& \text{outstanding-refs is-Write\_sb (takeWhile (Not \circ is-volatile-Write\_sb) sb)}_j = \\
& \{\} ;
\forall j < \text{length ts}. i \neq j \implies
\begin{align*}
& \text{let } (\cdot, \cdot, \cdot, \cdot, \cdot, \cdot) = ts!j \text{ in } a \notin \text{outstanding-refs is-Write\_sb (takeWhile (Not \circ is-volatile-Write\_sb) sb)}_j \\
& \implies
flush (\text{takeWhile (Not \circ is-volatile-Write\_sb) 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))
\end{align*}
\[\text{proof}\]

Lemma (in program)
\[\forall sb'. \text{history-consistent } \varnothing (hd-prog p (sb@sb')) (sb@sb') \implies
last-prog p (sb@sb') = p \implies
\text{last-prog (hd-prog p (sb@sb')) sb = hd-prog p sb'}
\[\text{proof}\]

Lemma last-prog-to-last-prog-same: \[\forall p'. \text{last-prog p'} sb = p \implies \text{last-prog p sb = p}\]
\[\text{proof}\]

Lemma last-prog-hd-prog-same: \[\text{last-prog p'} sb = p; \text{hd-prog p'} sb = p' \implies \text{hd-prog p sb = p'}\]
\[\text{proof}\]

Lemma last-prog-hd-prog-last-prog:
\[\text{last-prog p'} (sb@sb') = p \implies \text{hd-prog p'} (sb@sb') = p' \implies
\text{last-prog (hd-prog p sb')} sb = \text{last-prog p sb}
\[\text{proof}\]

Lemma (in program) last-prog-hd-prog-append:\n\[\forall sb'. \text{history-consistent } \varnothing (hd-prog p (sb@sb')) (sb@sb') \implies\]
\[\text{last-prog } p \ (\text{sb} @ \text{sb'}) = p \implies\]
\[\text{last-prog } (\text{hd-prog } p \ \text{sb'}) \ \text{sb} = \text{hd-prog } p \ \text{sb'}\]

\(\langle\text{proof}\rangle\)

\textbf{lemma} flush-all-until-volatile-write-Write\textsubscript{sb}-non-volatile-commute:
\[\forall i m. \ [i < \text{length } ts; ts!i=(p, is, xs, \text{Write}_{\text{sb}} \ False \ a \ sop \ v \ A \ L \ R \ W \# \text{sb}, D, O, R);\]
\[\forall i < \text{length } ts. \ (\forall j < \text{length } ts. \ i \neq j \implies\]
\[\text{let } (r, r, r, r, r, r) = ts!i;\]
\[\text{let } (r, r, r, r, r, r) = ts!j\]
\[\text{in outstanding-refs is-Write}_{\text{sb}} \text{ sb}_i \cap\]
\[\text{outstanding-refs is-Write}_{\text{sb}} \text{ (takeWhile (Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}_j) = \{\}];\]
\[\forall j < \text{length } ts. \ i \neq j \implies\]
\[\text{let } (r, r, r, r, r, r) = ts!j \text{ in a } \notin \text{ outstanding-refs is-Write}_{\text{sb}} \text{ (takeWhile (Not } \circ \text{is-volatile-Write}_{\text{sb}}) \text{ sb}_j)]\]
\[\implies \text{flush-all-until-volatile-write (ts}[i := (p, is, xs, \text{sb, D}', O, R')])(m(a := v)) = \text{flush-all-until-volatile-write ts m}\]

\(\langle\text{proof}\rangle\)

\textbf{lemma (in program) history-consistent-access-last-read'}:
\[\forall p. \text{history-consistent } \theta \ p (\text{sb } @ [\text{Read}_{\text{sb}} \text{ volatile a t v}]) \implies\]
\[\theta t = \text{Some } v\]

\(\langle\text{proof}\rangle\)

\textbf{lemma (in program) history-consistent-access-last-read}:
\[\text{history-consistent } \theta \ p \ (\text{rev (Read}_{\text{sb}} \text{ volatile a t v } \# \text{sb})) \implies \theta t = \text{Some } v\]

\(\langle\text{proof}\rangle\)

\textbf{lemma flush-all-until-volatile-write-Read\textsubscript{sb}-commute}:
\[\forall i m. \ [i < \text{length } ts; ts!i=(p, is, \theta, \text{Read}_{\text{sb}} \text{ volatile a t v } \# \text{sb}, D, O, R)];\]
\[\implies \text{flush-all-until-volatile-write (ts}[i := (p, is, \theta, \text{sb, D}', O, R')]) \text{ m} = \text{flush-all-until-volatile-write ts m}\]

\(\langle\text{proof}\rangle\)

\textbf{lemma flush-all-until-volatile-write-Ghost\textsubscript{sb}-commute}:
\[\forall i m. \ [i < \text{length } ts; ts!i=(p, is, \theta, \text{Ghost}_{\text{sb}} \text{ A L R W } \# \text{sb}, D, O, R)];\]
\[\implies \text{flush-all-until-volatile-write (ts}[i := (p', is', \theta', \text{sb, D}', O, R')]) \text{ m} = \text{flush-all-until-volatile-write ts m}\]

\(\langle\text{proof}\rangle\)

\textbf{lemma flush-all-until-volatile-write-Prog\textsubscript{sb}-commute}:
\[\forall i m. \ [i < \text{length } ts; ts!i=(p, is, \theta, \text{Prog}_{\text{sb}} p_1 p_2 \text{ mis } \# \text{sb}, D, O, R)];\]
\[\implies \text{flush-all-until-volatile-write (ts}[i := (p, is, \theta, \text{sb, D}', O, R')]) \text{ m} = \text{flush-all-until-volatile-write ts m}\]

\(\langle\text{proof}\rangle\)

\textbf{lemma flush-all-until-volatile-write-append-Prog\textsubscript{sb}-commute}:
\[\forall i m. \ [i < \text{length } ts; ts!i=(p, is, \theta, \text{sb, D}, O, R)];\]

\[\]
\[ \Longrightarrow \text{flush-all-until-volatile-write} \left( \text{ts}[i := (p_2, \text{is}@[\text{mis}], \emptyset, \text{sb}@[\text{Prog}_{sb} \ p_1 \ p_2 \ \text{mis}], O', R')] \right) \ m \]
\[ = \text{flush-all-until-volatile-write ts m} \]

**lemma** (in program) history-consistent-append-Prog\(_{sb}\):  
**assumes** step: \( \emptyset \vdash p \rightarrow_p (p', \text{mis}) \)  
**shows** history-consistent \( \emptyset \) \((\text{hd-prog} p \\ \text{xs}) \ \text{xs} \Longrightarrow \text{last-prog} p \\ \text{xs} = p \Longrightarrow \)
\[ \text{history-consistent} \emptyset \ (\text{hd-prog} p' (\text{xs}@[@\text{Prog}_{sb} \ p \ p' \ \text{mis}])) \ (\text{xs}@[@\text{Prog}_{sb} \ p \ p' \ \text{mis}]) \]

\[ \langle \text{proof} \rangle \]

**primrec** release :: \('\ a \ \text{memref list} \Rightarrow \text{addr set} \Rightarrow \text{rels} \Rightarrow \text{rels} \)
**where**
\[ \text{release} \ [\] \ S \ R = \ R \]
\[ | \text{release} \ (x#xs) \ S \ R = \]
\[ \text{(case } x \text{ of} \]
\[ \text{Write}_{sb} \ \text{volatile} - - - A \ L \ R \ W \Rightarrow \]
\[ \text{(if } \text{volatile} \text{ then release } \text{xs } (S \cup R - L) \ \text{Map.empty} \]
\[ \text{else release } \text{xs } S \ R) \]
\[ | \text{Ghost}_{sb} \ A \ L \ R \ W \Rightarrow \text{release } \text{xs } (S \cup R - L) \ (\text{augment-rels} S \ R \ R) \]
\[ | - \Rightarrow \text{release } \text{xs } S \ R) \]

**lemma** augment-rels-shared-exchange: \( \forall a \in R. (a \in S') = (a \in S) \Longrightarrow \text{augment-rels} S \ R \ R = \text{augment-rels} S' \ R \ R \)

\[ \langle \text{proof} \rangle \]

**lemma** sharing-consistent-shared-exchange:  
**assumes** shared-eq: \( \forall a \in \text{all-acquired sb}. S' a = S a \)  
**assumes** consis: sharing-consistent \( S \ O \ sb \)  
**shows** sharing-consistent \( S' \ O \ sb \)

\[ \langle \text{proof} \rangle \]

**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 \((\text{dom } S') \ R = \text{release sb } (\text{dom } S) \ R \)

\[ \langle \text{proof} \rangle \]

**lemma** release-append:
locale xvalid-program = valid-program +

fixes valid

assumes valid-implies-valid-prog:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}); \text{valid ts} \implies \text{valid-prog } p \]

assumes valid-implies-valid-prog-hd:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}); \text{valid ts} \implies \text{valid-prog } (\text{hd-prog } p \text{ sb}) \]

assumes distinct-load-tmps-prog-step:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}); \emptyset \vdash p \rightarrow p (p', is'); \text{valid ts} \]
\[ \implies \]  
\[ \text{distinct-load-tmps is'} \land \]  
\[ (\text{load-tmps is'} \cap \text{load-tmps is} = \{\}) \land \]  
\[ (\text{load-tmps is'} \cap \text{read-tmps sb}) = \{\} \]

assumes valid-data-dependency-prog-step:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}); \emptyset \vdash p \rightarrow p (p', is'); \text{valid ts} \]
\[ \implies \]  
\[ \text{data-dependency-consistent-instrs } (\text{dom } \emptyset \cup \text{load-tmps is}) \text{ is'} \land \]  
\[ \text{load-tmps is'} \cap \bigcup (\text{fst ' store-sops is}) = \{\} \land \]  
\[ \text{load-tmps is'} \cap \bigcup (\text{fst ' write-sops sb}) = \{\} \]

assumes load-tmps-fresh-prog-step:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}); \emptyset \vdash p \rightarrow p (p', is'); \text{valid ts} \]
\[ \implies \]  
\[ \text{load-tmps is'} \cap \text{dom } \emptyset = \{\} \]

assumes valid-sops-prog-step:
\[ \emptyset \vdash p \rightarrow p (p', is'); \text{valid-prog } p \implies \forall \text{sop} \in \text{store-sops is'} \text{. valid-sop } \text{sop} \]

assumes prog-step-preserves-valid:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}); \emptyset \vdash p \rightarrow p (p', is'); \text{valid ts} \implies \]  
\[ \text{valid } (\text{ts}[i := (p', is@is', \emptyset, sb@[\text{Prog sb p p'} is'], D, \mathcal{O}, \mathcal{R}])]) \]

assumes flush-step-preserves-valid:
\[ i < \text{length ts}; \]
\[ \text{ts}!i = (p, is, \emptyset, sb, D, \mathcal{O}, \mathcal{R}, S); (m, sb, \mathcal{O}, \mathcal{R}, S) \rightarrow f (m', sb', \mathcal{O}', \mathcal{R}', S'); \text{valid ts} \implies \]  
\[ \text{valid } (\text{ts}[i := (p, is, \emptyset, sb', D, \mathcal{O}', \mathcal{R}'])]) \]

assumes sbh-step-preserves-valid:
\[ i < \text{length ts}; \]
tsli = (p, is, sb, D, O, R);
(is, θ, sb, m, D, O, R, S) →_{sbh} (is′, θ′, sb′, m′, D′, O′, R′, S′);
valid ts]
⇒
valid (ts[i := (p, is′, θ′, sb′, D′, O′, R)])

lemma refl′: x = y ⇒ r^∗∗ x y
(proof)

lemma no-volatile-Read_{sb}-volatile-reads-consistent:
\[ ∀ m . \text{outstanding-refs is-volatile-Read}_{sb} \sb = \{ \} \implies \text{volatile-reads-consistent m sb} \]
(proof)

theorem (in program) flush-store-buffer-append:
shows \[ \{ i < \text{length ts};
\text{instrs} (\sb @ \sb′) \at \is_{sb} = \text{is} @ \text{prog-instrs} (\sb @ \sb′); \]
causal-program-history is_{sb} (sb@sb′);

\text{tsli} = (p, is, θ | (\text{dom } θ \setminus \text{read-tmps} (\sb @ \sb′)), x, D, O, R);

p = \text{hd-prog} \ p_{sb} (\sb @ \sb′);

(last-prog \ p_{sb} (\sb @ \sb′)) = \ p_{sb};

\text{reads-consistent True } O′ \ m \ sb;

\text{history-consistent } θ p (\sb @ \sb′);

\forall \text{sop} \in \text{write-sops sb}, \text{valid-sop sop};

\text{distinct-read-tmps} (\sb @ \sb′);

\text{volatile-reads-consistent m sb} \]
⇒
∃ is′, instrs sb′ @ is_{sb} = is′ @ prog-instrs sb′ ∧

\text{(ts, m, S) ⇒ d}^*

\text{(ts[i := (hd-prog (last-prog p_{sb} sb′) \ sb, is′, θ | (\text{dom } θ \setminus \text{read-tmps sb′}), x,}

\text{(D ∨ outstanding-refs is-volatile-Write}_{sb} \sb \neq \{ \}),

\text{acquired True sb O, release sb (dom S) R}],} \text{flush sb m, share sb S})
(proof)

corollary (in program) flush-store-buffer:
assumes i-bound: i < \text{length ts}
assumes instrs: instrs sb @ is_{sb} = is @ prog-instrs sb
assumes cph: causal-program-history is_{sb} sb
assumes ts-i: tsli = (p, is, θ | (\text{dom } θ \setminus \text{read-tmps sb}), x, D, O, R)
assumes p: p = \text{hd-prog} \ p_{sb} sb
assumes last-prog: (last-prog \ p_{sb} sb) = \ p_{sb}
assumes reads-consis: \text{reads-consistent True } O′ \ m \ sb
assumes hist-consis: \text{history-consistent } θ p \ sb
assumes valid-sops: ∀ \text{sop} \in \text{write-sops sb}, \text{valid-sop sop}
assumes dist: distinct-read-tmps sb
assumes vol-read-consis: volatile-reads-consistent m sb
shows (ts,m,S) ⇒ d*
  (ts[i]:=p_{sb}, is_{sb}, θ,x,
   D ∨ outstanding-refs is-volatile-Write_{sb} sb ≠ {}), acquired True sb O, release sb
   (dom S) R),
   flush sb m, share sb S)
⟨proof⟩

lemma last-prog-same-append: ∃ xs p_{sb}. last-prog p_{sb} (sb@xs) = p_{sb} =⇒ last-prog p_{sb} xs
= p_{sb}
⟨proof⟩

lemma reads-consistent-drop-volatile-writes-no-volatile-reads:
  ∃ pending-write O m. reads-consistent pending-write O m sb =⇒ outstanding-refs is-volatile-Read_{sb} ((dropWhile (Not ◦ is-volatile-Write_{sb})) sb) = {}
⟨proof⟩

lemma reads-consistent-flush-other:
  assumes no-volatile-Write_{sb}-sb: outstanding-refs is-volatile-Write_{sb} sb = {}
  shows ∃ m pending-write O.
   [outstanding-refs (Not ◦ is-volatile-Read_{sb}) xs ∩ outstanding-refs is-non-volatile-Write_{sb} sb = {}]:
     reads-consistent pending-write O m xs] =⇒ reads-consistent pending-write O (flush sb m) xs
⟨proof⟩

lemma reads-consistent-flush-independent:
  assumes no-volatile-Write_{sb}-sb: outstanding-refs is-Write_{sb} sb ∩ outstanding-refs
   is-non-volatile-Read_{sb} xs = {}
  assumes consis: reads-consistent pending-write O m xs
  shows reads-consistent pending-write O (flush sb m) xs
⟨proof⟩

lemma reads-consistent-flush-all-until-volatile-write-aux:
  assumes no-reads: outstanding-refs is-volatile-Read_{sb} xs = {}
  shows ∃ m pending-write O’. [reads-consistent pending-write O’ m xs; ∀ i < length ts.
   let (p,is,θ, sb, D,O,R) = ts!i in
   outstanding-refs (Not ◦ is-volatile-Read_{sb}) xs ∩ outstanding-refs is-non-volatile-Write_{sb} (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) = {}
] =⇒ reads-consistent pending-write O’ (flush-all-until-volatile-write ts m) xs
⟨proof⟩
lemma reads-consistent-flush-other':
assumes no-volatile-Write\_sb: outstanding-refs is-volatile-Write\_sb sb = {} 
shows \( \forall m. \ O \).
\[ \text{outstanding-refs is-non-volatile-Write}\_sb \sb \cap \]
\( \text{outstanding-refs is-volatile-Write}\_sb \xs \cup \)
\( \text{outstanding-refs is-non-volatile-Write}\_sb \xs \cup \)
\( \text{outstanding-refs is-non-volatile-Read}_\sb (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \xs) \)
\( \cup \)
\( \text{outstanding-refs is-non-volatile-Read}_\sb (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \xs) \)
\( \cup \)
\( (O \cup \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \xs)) \)
\) = \{\};
reads-consistent False O m xs;
read-only-reads O (takeWhile (Not \circ \text{is-volatile-Write}_\sb) \xs) \subseteq RO]
\[ \implies \text{reads-consistent False O (flush sb m) xs} \]
⟨proof⟩

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 \circ \text{is-volatile-Write}_\sb) \xs) \subseteq RO \)
sows \( \forall m. \ [\text{reads-consistent False O' m xs; } \forall i < \text{length ts.} \)
\[ \text{let } (p,\text{is},\theta,\sb,D,O) = ts!i \text{ in} \]
\( \text{outstanding-refs is-non-volatile-Write}_\sb (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \sb) \cap \)
\( \text{outstanding-refs is-volatile-Write}_\sb \xs \cup \)
\( \text{outstanding-refs is-non-volatile-Write}_\sb \xs \cup \)
\( \text{outstanding-refs is-non-volatile-Read}_\sb (\text{dropWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \xs) \)
\( \cup \)
\( \text{outstanding-refs is-non-volatile-Read}_\sb (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \xs) \)
\( \cup \)
\( (O' \cup \text{all-acquired} (\text{takeWhile} (\text{Not} \circ \text{is-volatile-Write}_\sb) \xs)) \)
\) = \{\}
\[ \implies \text{reads-consistent False O' (flush-all-until-volatile-write ts m) xs} \]
⟨proof⟩

lemma in-outstanding-refs-cases [consumes 1, case-names Write\_sb Read\_sb]:
a \in \text{outstanding-refs P }\xs \implies
\[ (\forall \text{volatile sop v A L R W. } \text{Write}\_sb \text{volatile a sop v A L R W) }\in \text{set xs }\implies P \]
(Write\_sb \text{volatile a sop v A L R W) }\implies C) \implies
\[ (\forall \text{volatile t v. } \text{Read}_\sb \text{volatile a t v) }\in \text{set xs }\implies P \]
(Read\_sb \text{volatile a t v) }\implies C)
\[ \Rightarrow C \]

\textit{proof}

\textbf{lemma} dropWhile-Cons: \((\text{dropWhile } P \; \text{x}s) = \text{x}\#\text{ys} \Rightarrow \neg P \; \text{x} \)

\textit{proof}

\textbf{lemma} reads-consistent-dropWhile:

\[
\text{reads-consistent pending-write } \mathcal{O} \; m \; (\text{dropWhile } (\neg \circ \text{is-volatile-Write}_s) \; sb) = \\
\text{reads-consistent True } \mathcal{O} \; m \; (\text{dropWhile } (\neg \circ \text{is-volatile-Write}_s) \; sb)
\]

\textit{proof}

\textbf{theorem}

\[
\text{reads-consistent-flush-all-until-volatile-write:} \\
\forall i \; m \; \text{pending-write.} \ [(\text{valid-ownership-and-sharing } \mathcal{S} \; \text{ts}; \\
i < \text{length ts; ts}!i = (p, \text{is}, \theta, \text{sb}; D, \mathcal{O}, R)]; \text{reads-consistent pending-write } \mathcal{O} \; m \; \text{sb} \] \\
\Rightarrow \text{reads-consistent True } (\text{acquired True } (\text{takeWhile } (\neg \circ \text{is-volatile-Write}_s) \; sb) \; \mathcal{O}) \\
(\text{flush-all-until-volatile-write ts } m) \; (\text{dropWhile } (\neg \circ \text{is-volatile-Write}_s) \; sb)
\]

\textit{proof}

\textbf{lemma} split-volatile-Write\_sb-in-outstanding-refs:

\[
a \in \text{outstanding-refs is-volatile-Write}_{sb} \; \text{x}s \Rightarrow (\exists \text{sop v ys zs A L R W} \; \text{x}s = \text{ys}(\text{Write}_{sb} \; \text{True} \; \text{sop v A L R W}#\text{zs}))
\]

\textit{proof}

\textbf{lemma} sharing-consistent-mono-shared:

\[
\forall \mathcal{S} \; \mathcal{S}' \; \mathcal{O}. \\
\text{dom } \mathcal{S} \subseteq \text{dom } \mathcal{S}' \Rightarrow \text{sharing-consistent } \mathcal{S} \; \mathcal{O} \; sb \Rightarrow \text{sharing-consistent } \mathcal{S}' \; \mathcal{O} \; sb
\]

\textit{proof}

\textbf{lemma} sharing-consistent-mono-owns:

\[
\forall \mathcal{O} \; \mathcal{O}' \; \mathcal{S}. \\
\mathcal{O} \subseteq \mathcal{O}' \Rightarrow \text{sharing-consistent } \mathcal{S} \; \mathcal{O} \; sb \Rightarrow \text{sharing-consistent } \mathcal{S} \; \mathcal{O}' \; sb
\]

\textit{proof}

\textbf{primrec} all-shared :: \( 'a \) memref list \Rightarrow addr set

\textbf{where}

\[
\text{all-shared } [] = \{\}
\]

\[
| \text{all-shared } (i\#\text{is}) = \\
\text{(case i of} \\
\text{Write}_{sb} \text{volatile } - - - A L R W \Rightarrow (\text{if volatile then } R \cup \text{all-shared is all-shared is}) \\
| \text{Ghost}_{sb} A L R W \Rightarrow R \cup \text{all-shared is} \\
| - \Rightarrow \text{all-shared is})
\]

\textbf{lemma} sharing-consistent-all-shared:

\[
\forall \mathcal{S} \; \mathcal{O}. \text{sharing-consistent } \mathcal{S} \; \mathcal{O} \; sb \Rightarrow \text{all-shared sb } \subseteq \text{dom } \mathcal{S} \cup \mathcal{O}
\]

100
lemma sharing-consistent-share-all-shared:
\[ \forall S. \text{dom (share } S) \subseteq \text{dom } S \cup \text{all-shared } S \]

primrec all-unshared :: 'a memref list \Rightarrow \text{addr set}
where
  all-unshared [] = {}
  | all-unshared (i#is) =
    (case i of
      Write sb volatile - - - A L R W \Rightarrow (if volatile then L \cup \text{all-unshared } is else all-unshared is)
    | Ghost sb A L R W \Rightarrow L \cup \text{all-unshared } is
    | - \Rightarrow \text{all-unshared } is)

lemma all-unshared-append: all-unshared (xs @ ys) = all-unshared xs \cup \text{all-unshared } ys

lemma freshly-shared-owned:
\[ \forall S O. \text{sharing-consistent } S O \text{ sb } \implies \text{dom (share } S) - \text{dom } S \subseteq O \]

lemma unshared-all-unshared:
\[ \forall S O. \text{sharing-consistent } S O \text{ sb } \implies \text{dom } S - \text{dom (share } S) \subseteq \text{all-unshared } sb \]

lemma unshared-acquired-or-owned:
\[ \forall S O. \text{sharing-consistent } S O \text{ sb } \implies \text{all-unshared } sb \subseteq \text{all-acquired } sb \cup O \]

lemma all-shared-acquired-or-owned:
\[ \forall S O. \text{sharing-consistent } S O \text{ sb } \implies \text{all-shared } sb \subseteq \text{all-acquired } sb \cup O \]

lemma sharing-consistent-preservation:
\[ \forall S S' O. \]
\[ \text{sharing-consistent } S O \text{ sb; all-acquired } sb \cap \text{dom } S - \text{dom } S' = \{\}; all-unshared } sb \cap \text{dom } S' - \text{dom } S = \{\} \]
\[ \implies \text{sharing-consistent } S' O \text{ sb} \]
∀\(i < \text{length ts}\). let (\(\_\_\_\_\text{sb}_{i}\_\_\_\_\)) = ts!i in

all-acquired \(\text{sb} \cap \text{dom } S - \text{dom } S' = \{\}\) \& all-unshared \(\text{sb} \cap \text{dom } S' - \text{dom } S = \{\}\)

shows sharing-consis \(S'\) ts

〈proof〉

lemma (in sharing-consis) sharing-consis-shared-exchange:

assumes dist:

∀\(i < \text{length ts}\). let (\(\_\_\_\_\text{sb}_{i}\_\_\_\_\)) = ts!i in

∀\(a \in \text{all-acquired sb}\). \(S' a = S a\)

shows sharing-consis \(S'\) ts

〈proof〉

lemma all-acquired-takeWhile: all-acquired (takeWhile \(P\) \(sb\)) \(\subseteq\) all-acquired \(sb\)

〈proof〉

lemma all-acquired-dropWhile: all-acquired (dropWhile \(P\) \(sb\)) \(\subseteq\) all-acquired \(sb\)

〈proof〉

lemma acquired-share-owns-shared:

assumes consis: sharing-consistent \(S\) \(O\) \(sb\)

shows acquired pending-write \(sb\) \(O \cup \text{dom}\) (share \(sb\) \(S\)) \(\subseteq\) \(O \cup \text{dom} S\)

〈proof〉

lemma acquired-owns-shared:

assumes consis: sharing-consistent \(S\) \(O\) \(sb\)

shows acquired True \(sb\) \(O \subseteq\) \(O \cup \text{dom} S\)

〈proof〉

lemma share-owns-shared:

assumes consis: sharing-consistent \(S\) \(O\) \(sb\)

shows dom (share \(sb\) \(S\)) \(\subseteq\) \(O \cup \text{dom} S\)

〈proof〉

lemma all-shared-append: all-shared (\(xs@ys\)) = all-shared \(xs\) \& all-shared \(ys\)

〈proof〉

lemma acquired-union-notin-first:

\(\forall \text{pending-write } A B.\ a \in \text{acquired pending-write } sb\ (A \cup B) \implies a \notin A \implies a \in \text{acquired pending-write } sb\ B\)

〈proof〉
lemma split-all-acquired-in:
a ∈ all-acquired xs \implies
(\exists \text{sop } a' \text{ v } ys \text{ z } s \text{ a L R W. } xs = ys @ \text{Write}_{sb} \text{ True } a' \text{ sop } v \text{ A L R W# } zs \land a \in A) \lor
(\exists A \text{ L R W } ys \text{ z } s. \text{ xs } = ys @ \text{Ghost}_{sb} \text{ A L R W# } zs \land a \in A)
\langle proof \rangle

lemma split-Write_{sb}-in-outstanding-refs:
a ∈ outstanding-refs is-Write_{sb} xs \implies (\exists \text{sop volatile } v \text{ z } s \text{ A L R W. } xs = ys @(\text{Write}_{sb} \text{ volatile } a \text{ sop } v \text{ A L R W# } zs))
\langle proof \rangle

lemma outstanding-refs-is-Write_{sb}-union:
outstanding-refs is-Write_{sb} xs =
(outstanding-refs is-volatile-Write_{sb} xs \cup outstanding-refs is-non-volatile-Write_{sb} xs)
\langle proof \rangle

lemma rtranclp-r-rtranclp: 
[r** x y; r y z] \implies r** x z
\langle proof \rangle

lemma r-rtranclp-rtranclp: [r x y; r** y z] \implies r** x z
\langle proof \rangle

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
\langle proof \rangle

lemma outstanding-non-volatile-Read_{sb}-acquired-or-read-only-reads:
\bigwedge O S \text{ pending-write.}
[non-volatile-owned-or-read-only pending-write S O sb; a \in outstanding-refs is-non-volatile-Read_{sb} sb]
\implies a \in acquired-reads True sb O \lor a \in read-only-reads O sb
\langle proof \rangle

lemma acquired-reads-union: \bigwedge \text{pending-writes } A B.
[a \in acquired-reads pending-writes sb (A \cup B); a \notin A] \implies a \in acquired-reads pending-writes sb B
\langle proof \rangle

lemma non-volatile-writes-unshared-no-outstanding-non-volatile-Write_{sb}: \bigwedge S S'.
[non-volatile-writes-unshared S sb;
\forall a \in dom S' - dom S. a \notin outstanding-refs is-non-volatile-Write_{sb} sb ]
\implies non-volatile-writes-unshared S' sb
\langle proof \rangle
**Theorem** sharing-consis-share-all-until-volatile-write:
\[ \forall S \text{ ts}' [\text{ownership-distinct ts; sharing-consis } S \text{ ts; length ts}' = \text{length ts; } \forall i < \text{length ts}. \]
\[
\begin{align*}
&\text{(let } (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \text{ts}!i; \\
&(\cdot,\cdot,\cdot,\cdot',\cdot,\cdot) = \text{ts}'!i \\
in \text{O}' = \text{acquired True (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) \text{ sb)} \text{ O } \land \\
\text{sb}' = \text{dropWhile (Not } \circ \text{is-volatile-Write}_{sb} ) \text{ sb}] \implies \\
\text{sharing-consis (share-all-until-volatile-write ts } S \text{) ts}' \land \\
\text{dom (share-all-until-volatile-write ts } S \text{) - dom } S \subseteq \\
\bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{ O) } \setminus \text{ set ts}) \land \\
\text{dom } S - \text{ dom (share-all-until-volatile-write ts } S \text{) } \subseteq \\
\bigcup ((\lambda(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot). \text{ all-acquired sb } \cup \text{ O}) \setminus \text{ set ts})
\end{align*}
\]
\langle proof \rangle

**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, is, θ, sb, D, O, R)
**Shows** sharing-consistent (share-all-until-volatile-write ts S)
\[
\begin{align*}
&\text{(acquired True (takeWhile (Not } \circ \text{is-volatile-Write}_{sb} ) \text{ sb)} \text{ O}) \\
&\text{(dropWhile (Not } \circ \text{is-volatile-Write}_{sb} ) \text{ sb)}
\end{align*}
\]
\langle proof \rangle

**Lemma** restrict-map-UNIV [simp]: S \models ^\uparrow\text{ UNIV} = S
\langle proof \rangle

**Lemma** share-all-until-volatile-write-Read-commute:
\[ \forall S \text{ i. } [\text{i < length ts; ls}!i=(p, \text{Read volatile a } t\#\text{is,θ, sb, D, O})]
\]
\[ \implies \\
\text{share-all-until-volatile-write}
\]
\[
\begin{align*}
&(\text{ls}[i := (p, \text{is}, \theta (t\mapsto v), \text{sb } @ \text{[Read}_{sb} \text{ volatile a } t v], D', O)]) S = \\
&\text{share-all-until-volatile-write ls S}
\end{align*}
\]
\langle proof \rangle

**Lemma** share-all-until-volatile-write-Write-commute:
\[ \forall S \text{ i. } [\text{i < length ts; ls}!i=(p, \text{Write volatile a } (D,f) A L R W\#\text{is,θ, sb, D, O})]
\]
\[ \implies \\
\text{share-all-until-volatile-write}
\]
\[
\begin{align*}
&(\text{ls}[i := (p, \text{is}, \theta, \text{sb } @ \text{[Write}_{sb} \text{ volatile a } t (f \theta) A L R W], D', O)]) S = 
\end{align*}
\]
104
share-all-until-volatile-write ls S

\text{lemma} \ share-all-until-volatile-write-RMW-commute:
\text{shows} \ \forall \ i. \ \left[ \left[ i < \text{length} \ ls; \ ls!i=(p,\text{RMW} a t (D,f) \ \text{cond ret} \ A \ L \ R \ W\#is,\emptyset,[]),D,O \right] \right] \ \implies \ \text{share-all-until-volatile-write} (ls[i := (p',is,\emptyset',[]),D,O']) S = \text{share-all-until-volatile-write} ls S

\text{lemma} \ share-all-until-volatile-write-Fence-commute:
\text{shows} \ \forall \ i. \ \left[ \left[ i < \text{length} \ ls; \ ls!i=(p,\text{Fence}#is,\emptyset,[]),D,O,R \right] \right] \ \implies \ \text{share-all-until-volatile-write} (ls[i := (p,is,\emptyset,[]),D,O,R']) S = \text{share-all-until-volatile-write} ls S

\text{lemma} \ unshared-share-in: \ \forall S. \ a \in \text{dom} \ S \implies a \notin \text{all-unshared} sb \implies a \in \text{dom} (\text{share sb} S)

\text{lemma} \ dom-eq-dom-share-eq: \ \forall S S'. \ \text{dom} S = \text{dom} S' \implies \text{dom} (\text{share sb} S) = \text{dom} (\text{share sb} S')

\text{lemma} \ share-union:
\forall A B. \ \left[ a \in \text{dom} (\text{share sb} (A \oplus z \ B)); \ a \notin \text{dom} A \right] \implies a \in \text{dom} (\text{share sb} (\text{Map.empty} \oplus z \ B))

\text{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})
**Lemma** dom-augment-rels-shared-eq: $\text{dom} \ (\text{augment-rels} \ S \ R \ \mathcal{R}) = \text{dom} \ (\text{augment-rels} \ S' \ R \ \mathcal{R})$

(proof)

**Lemma** dom-eq-SomeD1: $m = n \implies m \ x = \text{Some } y \implies n \ x \neq \text{None}$

(proof)

**Lemma** dom-eq-SomeD2: $m = n \implies n \ x = \text{Some } y \implies m \ x \neq \text{None}$

(proof)

**Lemma** dom-augment-rels-rels-eq: $\text{dom} \ \mathcal{R}' = \text{dom} \ \mathcal{R} \implies \text{dom} \ (\text{augment-rels} \ S \ R \ \mathcal{R}') = \text{dom} \ (\text{augment-rels} \ S \ R \ \mathcal{R})$

(proof)

**Lemma** dom-release-rels-eq: $\forall S \mathcal{R} \mathcal{R}'. \text{dom} \ \mathcal{R}' = \text{dom} \ \mathcal{R} \implies \text{dom} \ (\text{release} \ sb \ S \mathcal{R}') = \text{dom} \ (\text{release} \ sb \ S \mathcal{R})$

(proof)

**Lemma** dom-release-shared-eq: $\forall S S' \mathcal{R}. \text{dom} \ (\text{release} \ sb \ S' \mathcal{R}) = \text{dom} \ (\text{release} \ sb \ S \mathcal{R})$

(proof)

**Lemma** share-other-untouched:

$\forall \mathcal{O} S. \text{sharing-consistent} \ S \mathcal{O} \ sb \implies a \notin \mathcal{O} \cup \text{all-acquired} \ sb \implies \text{share} \ S \ a = S \ a$

(proof)

**Lemma** shared-owned:

$\forall \mathcal{O} S. \text{sharing-consistent} \ S \mathcal{O} \ sb \implies a \notin \text{dom} \ S \implies a \in \text{dom} \ (\text{share} \ sb \ S) \implies a \in \mathcal{O} \cup \text{all-acquired} \ sb$

(proof)

**Lemma** share-all-shared-in: $a \in \text{dom} \ (\text{share} \ sb \ S) \implies a \in \text{dom} \ S \lor a \in \text{all-shared} \ sb$

(proof)

**Lemma** share-all-until-volatile-write-unowned:

**assumes** dist: ownership-distinct ts

**assumes** consis: sharing-consis S ts

**assumes** other: $\forall i \ p \ \text{is } \emptyset \ sb \ D \mathcal{O} \mathcal{R}. \ i < \text{length} \ ts \implies \text{tsli} = (p, \text{is}, \emptyset, \text{sb}, \mathcal{D}, \mathcal{O}, \mathcal{R}) \implies a \notin \mathcal{O} \cup \text{all-acquired} \ sb$

**shows** share-all-until-volatile-write ts S a = S a

(proof)

**Lemma** share-shared-eq: $\forall S' S. S' a = S a \implies \text{share} \ sb \ S' a = \text{share} \ sb \ S a$

(proof)
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 = \( (p, \text{is}, 0, \text{sb}, D, O, R) \)
assumes a-owned: \( a \in O \cup \) all-acquired sb
shows share-all-until-volatile-write ts \( S \) a = share (takeWhile (Not \circ \) is-volatile-Write\(_{sb}\) sb) \( S \) a
⟨proof⟩

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 = \( (p, \text{is}, 0, \text{sb}, D, O, R) \)
assumes a-owned: \( a \in O \cup \) all-acquired sb
shows share (dropWhile (Not \circ \) is-volatile-Write\(_{sb}\) sb) (share-all-until-volatile-write ts \( S \) a = share sb \( S \) a
⟨proof⟩

lemma (in ownership-distinct) in-shared-sb-share-all-until-volatile-write:
assumes consis: sharing-consis \( S \) ts
assumes i-bound: \( i < \) length ts
assumes ts-i: ts!i = \( (p, \text{is}, 0, \text{sb}, D, O, R) \)
assumes a-owned: \( a \in O \cup \) all-acquired sb
assumes a-share: \( a \in \text{dom} (share sb) \( S \) \)
shows \( a \in \text{dom} (\text{share (dropWhile (Not \circ \) is-volatile-Write\(_{sb}\) sb)} (\text{share-all-until-volatile-write ts} \( S \)))\)
⟨proof⟩

lemma owns-unshared-share-acquired:
\( \bigwedge S \ O. [\text{sharing-consistent } S \ O \ \text{sb; } a \in O; a \notin \text{all-unshared sb}] \)
\( \implies a \in \text{dom (share sb } S) \cup \text{acquired True sb } O \)
⟨proof⟩

lemma shared-share-acquired: \( \bigwedge S \ O. \text{sharing-consistent } S \ O \ \text{sb } \implies \)
\( a \in \text{dom } S \implies a \in \text{dom (share sb } S) \cup \text{acquired True sb } O \)
⟨proof⟩

lemma dom-release-takeWhile:
\( \bigwedge S \ R. \)
\( \text{dom (release (takeWhile (Not \circ \) is-volatile-Write\(_{sb}\) sb)} S \ R) = \)
\( \text{dom } R \cup \text{all-shared (takeWhile (Not \circ \) is-volatile-Write\(_{sb}\) sb)} \)
⟨proof⟩

lemma share-all-until-volatile-write-share-acquired:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis \( S \) ts

107
assumes a-notin: \( a \notin \text{dom } S \)
assumes a-in: \( a \in \text{dom } (\text{share-all-until-volatile-write } ts \ S) \)
shows \( \exists i < \text{length ts}. \)
let \((\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \text{ts}!i \)
in \( a \in \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb) \)
⟨proof⟩

lemma all-shared-share-acquired: \( \forall S \ O. \ \text{sharing-consistent } S \ O \ sb \implies \)
\( a \in \text{all-shared sb } \implies a \in \text{dom } (\text{share sb } S) \cup \text{acquired True sb } O \)
⟨proof⟩

lemma (in ownership-distinct) share-all-until-volatile-write-share-acquired:
assumes consis: \( \text{sharing-consis } S \ ts \)
assumes i-bound: \( i < \text{length ts} \)
assumes ts-i: \( \text{ts}!i = (p,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) \)
assumes a-in: \( a \in \text{dom } (\text{share-all-until-volatile-write } ts \ S) \)
shows \( a \in \text{dom } (\text{share sb } S) \land a \in \text{acquired True sb } O \land \leftrightharpoons \)
\( (\exists j < \text{length ts}. \ j \neq i \land \)
\( (\text{let } (\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) = \text{ts}!j \)
in \( a \in \text{all-shared } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}_{sb}) \ sb_j) ) \))
⟨proof⟩

lemma acquired-all-shared-in:
\( \forall A. \ a \in \text{acquired True sb } A \implies a \in \text{acquired True sb } \{ \} \lor (a \in A \land a \notin \text{all-shared sb}) \)
⟨proof⟩

lemma all-shared-acquired-in: \( \forall A. \ a \in A \implies a \notin \text{all-shared sb } \implies a \in \text{acquired True sb } A \)
⟨proof⟩

lemma owned-share-acquired: \( \forall S \ O. \ \text{sharing-consistent } S \ O \ sb \implies \)
\( a \in O \implies a \in \text{dom } (\text{share sb } S) \cup \text{acquired True sb } O \)
⟨proof⟩

lemma outstanding-refs-non-volatile-Read_{sb}-all-acquired:
\( \forall m S \ O \ \text{pending-write}. \)
\( [\text{reads-consistent pending-write } O \ m \ sb; \text{non-volatile-owned-or-read-only pending-write } S \ O \ sb; \)
a \in \text{outstanding-refs is-non-volatile-Read_{sb} sb} \]
\( \implies a \in O \lor a \in \text{all-acquired sb} \lor \)
a \in \text{read-only-reads O sb} \)
⟨proof⟩

108
lemma outstanding-refs-non-volatile-Read_{sb}-all-acquired-dropWhile:
assumes conis: reads-consistent pending-write \( O \) m sb
assumes nvo: non-volatile-owned-or-read-only pending-write \( S O \) sb
assumes out: \( a \in \) outstanding-refs is-non-volatile-Read_{sb} (dropWhile (Not o is-volatile-Write_{sb}) sb)
shows a \( \in O \lor a \in \) all-acquired sb \( \lor \)
a \( \in \) read-only-reads \( O \) sb
\( \langle proof \rangle \)
lemma share-commute:
\[ L R S O. \] [sharing-consistent \( S O \) sb;]
all-shared sb \( \cap \) L = \{\}; all-shared sb \( \cap \) A = \{\}; all-acquired sb \( \cap \) R = \{\};
all-unshared sb \( \cap \) R = \{\}; all-shared sb \( \cap \) R = \{\} \implies
\( \langle proof \rangle \)
lemma share-append-Ghost_{sb}:
\[ S R L. \] [ownership-distinct ts; sharing-consis \( S \) ts;]
\( \forall i p \in O R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) L = \{\};
\( \forall i p \in O R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) A = \{\};
\( \forall i p \in O R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\};
\( \forall i p \in O R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\};
\( \forall i p \in O R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\} \]\( \implies \)
share-all-until-volatile-write ts \( S \oplus W R \ominus A L \) = share-all-until-volatile-write ts \( S \oplus W R \ominus A L \)
\( \langle proof \rangle \)
lemma share-all-until-volatile-write-append-Ghost_{sb}:
\[ S R L. \] [ownership-distinct ts; sharing-consis \( S \) ts;]
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) L = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) A = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\} \]\( \implies \)
share-all-until-volatile-write ts \( S \oplus W R \ominus A L \) = share-all-until-volatile-write ts \( (S \oplus W R \ominus A L) \)
\( \langle proof \rangle \)
lemma share-append-Ghost_{sb}':
\[ S R L. \] [ownership-distinct ts; sharing-consis \( S \) ts;]
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) L = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) A = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-acquired (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-unshared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\};
\( \forall i p \in S R D \not\in sb. i < \) length ts \( \rightarrow tsli=(p,i,s,\varnothing,sb,D,O,R) \rightarrow \)
all-shared (takeWhile (Not o is-volatile-Write_{sb}) sb) \( \cap \) R = \{\} \]\( \implies \)
share-all-until-volatile-write ts \( S \oplus W R \ominus A L \) = share-all-until-volatile-append-Ghost_{sb}'}:
assumes no-out-VWrite\(_{sb}\): outstanding-refs is-volatile-Write\(_{sb}\) sb = {}

shows \(\bigwedge S\ i.\ [\text{ownership-distinct ts}; \text{sharing-consis } S\ ts];\)

\[ i < \text{length ts};\ \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R});\]
\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-shared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap L = {}
\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-shared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap A = {}
\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-unshared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap R = {}
\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-shared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap \text{all-unshared} (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap L = {}

\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-shared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap \text{all-unshared} (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap A = {}

\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-unshared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap \text{all-unshared} (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap R = {}

\[ \forall j\ p\ is\ R\ \text{\(\partial\)}\ sb.\ j < \text{length ts} \implies i \neq j \implies \text{tsli} = (p,\text{is},\text{\(\partial\)}\sb,\text{D},\text{O},\text{R}) \implies \]
all-shared (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap \text{all-unshared} (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap \text{all-unshared} (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap \text{all-unshared} (takeWhile (Not is-volatile-Write\(_{sb}\)) sb) \cap R = {}

\[ \implies \] share-all-until-volatile-write (ts\[i := (p',\text{is}',\partial',\sb, [\text{Ghost sb A L R W}], D', O'])\] S
\[ = \text{share-all-until-volatile-write ts S} \oplus W \ominus R \ominus A L\]

⟨proof⟩

lemma share-domain-changes:
\[ \bigwedge S S'.\ a \in \text{all-shared sb} \cup \text{all-unshared sb} \implies \text{share sb S'} a = \text{share sb S a} \]
⟨proof⟩

lemma share-domain-changesX:
\[ \bigwedge S S' X.\ \forall a \in X.\ S' a = S a \]
\[ \implies a \in \text{all-shared sb} \cup \text{all-unshared sb} \cup X \implies \text{share sb S'} a = \text{share sb S a} \]
⟨proof⟩

lemma share-unchanged:
\[ \bigwedge S.\ a \notin \text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb} \implies \text{share sb S a} = S a \]
⟨proof⟩

lemma share-augment-release-commute:
assumes dist: (R \cup L \cup A) \cap (\text{all-shared sb} \cup \text{all-unshared sb} \cup \text{all-acquired sb}) = {}

shows (share sb S \oplus W R \ominus A L) = \text{share sb (S \oplus W R \ominus A L)}
⟨proof⟩

lemma share-append-commute:
\[ \bigwedge y S.\ (\text{all-shared xs} \cup \text{all-unshared xs} \cup \text{all-acquired xs}) \cap \]
(\text{all-shared ys} \cup \text{all-unshared ys} \cup \text{all-acquired ys}) = {}
\[ \implies \text{share xs (share ys S)} = \text{share ys (share xs S)} \]
⟨proof⟩

lemma share-append-commute’:
assumes dist: (\text{all-shared xs} \cup \text{all-unshared xs} \cup \text{all-acquired xs}) \cap
shows \( \text{share} \ (\text{ys}@\text{xs}) \ S = \text{share} \ (\text{xs}@\text{ys}) \ S \)

\[\text{lemma} \ \text{share-all-until-volatile-write-share-commute:} \]
\[\text{shows} \ \bigwedge S \ ((\text{sb}:::\text{'a memref list})). \ \big[ \ \text{ownership-distinct ts; sharing-consis} \ S \ \text{ts; } \forall \ i \ p \ \text{is} \ \text{ORD} \ \theta \ \text{sb}::\text{'a memref list}. \ i < \text{length ts} \]
\[\rightarrow ts!i=(p,\text{is},\theta,\text{sb},\text{D},\text{O},\text{R}) \rightarrow \]
\[\text{(all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } \cup \]
\[\text{all-unshared (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } \cup \]
\[\text{all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } \cap \]
\[\text{(all-shared sb'} \cup \text{all-unshared sb'} \cup \text{all-acquired sb'}) = \{\} \]
\[\Rightarrow \]
\[\text{share-all-until-volatile-write} \ \text{ts} \ (\text{share} \ \text{sb}' \ S) = \]
\[\text{share} \ \text{sb'} \ (\text{share-all-until-volatile-write} \ \text{ts} \ S) \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{all-shared-takeWhile-subset: all-shared (takeWhile P sb) } \subseteq \text{all-shared sb} \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{all-shared-dropWhile-subset: all-shared (dropWhile P sb) } \subseteq \text{all-shared sb} \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{all-unshared-takeWhile-subset: all-unshared (takeWhile P sb) } \subseteq \text{all-unshared sb} \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{all-unshared-dropWhile-subset: all-unshared (dropWhile P sb) } \subseteq \text{all-unshared sb} \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{all-acquired-takeWhile-subset: all-acquired (takeWhile P sb) } \subseteq \text{all-acquired sb} \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{all-acquired-dropWhile-subset: all-acquired (dropWhile P sb) } \subseteq \text{all-acquired sb} \]
\[\langle \text{proof} \rangle \]

\[\text{lemma} \ \text{share-all-until-volatile-write-flush-commute:} \]
\[\text{assumes} \ \text{takeWhile-empty: (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } = \[\]
\[\text{shows} \ \bigwedge S R L W A i. \ \big[ \ \text{ownership-distinct ts; sharing-consis} \ S \ \text{ts; } \forall \ i \ p \ \text{is} \ \text{ORD} \ \theta \ \text{sb}::\text{'a memref list}. \ i < \text{length ts} \]
\[\rightarrow ts!i=(p,\text{is},\theta,\text{sb},\text{D},\text{O},\text{R}) \rightarrow \]
\[\text{(all-shared (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } \cup \]
\[\text{all-unshared (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } \cup \]
\[\text{all-acquired (takeWhile (Not \circ \text{is-volatile-Write}_{sb} \) sb) } \cap \]
\[\text{(all-shared sb'} \cup \text{all-unshared sb'} \cup \text{all-acquired sb'}) = \{\} \]
\[\forall \ j \ p \ \text{is} \ \text{ORD} \ \theta \ \text{sb}::\text{'a memref list}. \ j < \text{length ts} \rightarrow i \neq j \]
\[\rightarrow ts!j=(p,\text{is},\theta,\text{sb},\text{D},\text{O},\text{R}) \rightarrow \]
\[\text{(all-shared sb } \cup \text{all-unshared sb } \cup \text{all-acquired sb) } \cap \]
\[\text{(R } \cup \text{ L } \cup \text{ A) } = \{\}] \]
share-all-until-volatile-write \((\text{ts[i := (p', is', \theta', \text{sb'}, D', O', R')]})\) \((S \oplus W R \ominus A L) = \) 
share \((\text{takeWhile (Not o is-volatile-Write}_{sb} \text{ sb'})\) \((\text{share-all-until-volatile-write ts S} \oplus W R \ominus A L)\) 
\langle proof \rangle

\text{lemma} \ share-all-until-volatile-write-Ghost_{sb}-commute:
\text{shows} \ \land S i. \ [[\text{ownership-distinct ts; sharing-consis S ts; } i < \text{ length ts; } \\
\text{ts[i]} = (p, is, \theta, \text{Ghost}_{sb} A L R W # sb, D, O, R)]], \forall j p is O D \theta \text{ sb. } j < \text{ length ts } \rightarrow i \neq j \rightarrow \text{ts[j]} = (p, is, \theta, \text{sb}, D, O, R) \rightarrow 
(\text{all-shared (takeWhile (Not o is-volatile-Write}_{sb} \text{ sb}) } \cup \text{ all-unshared (takeWhile (Not o is-volatile-Write}_{sb} \text{ sb})}) \cap 
(R \cup L \cup A) = \{\} \]

\Rightarrow \text{share-all-until-volatile-write \((\text{ts[i := (p', is', \theta', \text{sb'}, D', O', R')]})\) \((S \oplus W R \ominus A L) = \text{share-all-until-volatile-write ts S} \langle proof \rangle\)

\text{lemma} \ share-all-until-volatile-write-update-sb:
\text{assumes} \ congr: \ \land S i. \ [[i < \text{ length ts; } \text{ts[i]} = (p, is, \theta, \text{sb}, D, O, R)]]
\text{shows} \ \land S i. \ [[i < \text{ length ts; } \text{ts[i]} = (p, is, \theta, \text{sb}, D, O, R)]]
\Rightarrow \text{share-all-until-volatile-write ts S} = \text{share-all-until-volatile-write \((\text{ts[i := (p', is', \theta', \text{sb'}, D', O', R')]})\) S} \langle proof \rangle

\text{lemma} \ share-all-until-volatile-write-append-Ghost_{sb}':
\text{assumes} \ out-VWrite_{sb}: \text{outstanding-refs is-volatile-Write}_{sb} \text{ sb} \neq \{\}
\text{assumes} \ i-bound: i < \text{ length ts}
\text{assumes} \ ts-i: \ \text{ts[i]} = (p, is, \theta, \text{sb}, D, O, R)
\text{shows} \ \land S \ [[\text{share-all-until-volatile-write ts S} = \text{share-all-until-volatile-write \((\text{ts[i := (p', is', \theta', sb', D', O', R')]})\) S}]] \langle proof \rangle

\text{lemma} \ acquired-append-Prog_{sb}:
\land S. \ (\text{acquired pending-write (takeWhile (Not o is-volatile-Write}_{sb} \text{ sb}) (sb @ [Prog}_{sb} p1 p2 mis]))) S) = 
\text{acquired pending-write (takeWhile (Not o is-volatile-Write}_{sb} \text{ sb}) S} \langle proof \rangle

\text{lemma} \ outstanding-refs-non-empty-dropWhile:
\text{outstanding-refs P xs \neq \{\} } \Rightarrow \text{outstanding-refs P (dropWhile (Not o P) xs) \neq \{\} } \langle proof \rangle

\text{lemma} \ ex-not: \ \text{Ex Not}
lemma (in computation) concurrent-step-append:
  assumes step: (ts,m,S) ⇒ (ts',m',S')
  shows (xs@ts,m,S) ⇒ (xs@ts',m',S')
⟨proof⟩

primrec weak-sharing-consistent:: owns ⇒ 'a memref list ⇒ bool
where
weak-sharing-consistent O [] = True
| weak-sharing-consistent O (r#rs) =
  (case r of
    Write sb volatile - - A L R W ⇒
    (if volatile then L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ O ∧
      weak-sharing-consistent (O ∪ A − R) rs
    else weak-sharing-consistent O rs)
  | Ghost sb A L R W ⇒
    (L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ O ∧ weak-sharing-consistent (O ∪ A − R) rs
  | - ⇒ weak-sharing-consistent O rs)

lemma sharing-consistent-weak-sharing-consistent:
  \( ∧S O . \) sharing-consistent S O sb ⇒ weak-sharing-consistent O sb
⟨proof⟩

lemma weak-sharing-consistent-append:
  \( ∧O . \) weak-sharing-consistent O (xs @ ys) =
  (weak-sharing-consistent O xs ∧ weak-sharing-consistent (acquired True xs O) ys)
⟨proof⟩

lemma read-only-share-unowned: \( ∧O S . \)
  [weak-sharing-consistent O sb; a \notin O ∪ all-acquired sb; a ∈ read-only (share sb S)]
  ⇒ a ∈ read-only S
⟨proof⟩
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')
⟨proof⟩

lemma read-only-unacquired-share:

$\forall S \ O. \ ([O \cap \text{read-only} \ S = \emptyset]; \ \text{weak-sharing-consistent} \ O \ sb; \ a \in \text{read-only} \ S; \ a \not\in \text{all-acquired} \ sb ]$

$\implies a \in \text{read-only} (\text{share} \ sb \ S)$
⟨proof⟩

lemma read-only-share-unacquired:

$\forall \ O \ S. \ ([O \cap \text{read-only} \ S = \emptyset]; \ \text{weak-sharing-consistent} \ O \ sb; \ a \in \text{read-only} \ S; \ a \not\in \text{all-acquired} \ sb ]$

$\implies a \in \text{read-only} (\text{share} \ sb \ S) = \not\in \text{acquired True} \ sb \ O$
⟨proof⟩

lemma read-only-share-all-acquired-in:

$\forall S \ O. \ ([O \cap \text{read-only} \ S = \emptyset]; \ \text{weak-sharing-consistent} \ O \ sb; \ a \in \text{read-only} \ (\text{share} \ sb \ S))$

$\implies a \in \text{read-only} (\text{share} \ sb \ \text{Map.empty}) \lor (a \in \text{read-only} \ S \land a \not\in \text{all-acquired} \ sb)$
⟨proof⟩

lemma weak-sharing-consistent-preserves-distinct:

$\forall O \ S. \ \text{weak-sharing-consistent} \ O \ sb \implies O \cap \text{read-only} \ S = \emptyset \implies$

acquired True \ sb \ O \cap \text{read-only} (\text{share} \ sb \ S) = \emptyset$
⟨proof⟩

locale weak-sharing-consis =

fixes ts::("p",p store-buffer,boo,owns,rels) thread-config list

assumes weak-sharing-consis:

$\forall i \ p \ is \ O \ R \ D \ \emptyset \ sb.$

$[i < \text{length} \ ts; \ ts!is = (p, \text{is}, \emptyset, sb, D, O, R)]$

$\implies \ \text{weak-sharing-consistent} \ O \ sb$

sublocale sharing-consis ⊆ weak-sharing-consis
⟨proof⟩

lemma weak-sharing-consis-tl: weak-sharing-consis (t#ts) ⇒ weak-sharing-consis ts
⟨proof⟩
lemma read-only-share-all-until-volatile-write-unacquired:
\( \forall S. \) [ownership-distinct ts; read-only-unowned S ts; weak-sharing-consis ts; \\
\forall i < \text{length ts}. \) (let \((-,-,-,sb,-,O,R) = \text{ts!i}\) in \\
a \notin \text{all-acquired (takeWhile (Not o is-volatile-Write sb) sb));} \\
a \in \text{read-only S}] \\
\implies a \in \text{read-only (share-all-until-volatile-write ts S)}
\langle proof \rangle

lemma read-only-share-unowned-in:
[weak-sharing-consistent O sb; a \in \text{read-only (share sb S)}] \\
\implies a \in \text{read-only S} \cup O \cup \text{all-acquired sb}
\langle proof \rangle

lemma read-only-shared-all-until-volatile-write-subset:
\( \forall S. \) [ownership-distinct ts; weak-sharing-consis ts; \\
i < \text{length ts}; ts!i = (p,is,\theta,sb,D,O,R)] \\
\implies \text{acquired True (takeWhile (Not o is-volatile-Write sb) sb)} \cap \\
\text{read-only (share-all-until-volatile-write ts S)} = \{\}
\langle proof \rangle

lemma weak-sharing-consistent-preserves-distinct-share-all-until-volatile-write:
\( \forall S i. \) [ownership-distinct ts; read-only-unowned S ts;weak-sharing-consis ts; \\
i < \text{length ts} ts!i = (p,is,\theta,sb,D,O,R)] \\
\implies \text{acquired True (takeWhile (Not o is-volatile-Write sb) sb)} \cap \\
\text{read-only (share-all-until-volatile-write ts S)} = \{\}
\langle proof \rangle

lemma in-read-only-share-all-until-volatile-write:
assumes dist: ownership-distinct ts 
assumes consis: sharing-consis S ts 
assumes ro-unowned: read-only-unowned S ts 
assumes i-bound: i < \text{length ts} 
assumes ts-i: ts!i = (p,is,\theta,sb,D,O,R) 
assumes a-unacquired-others: \forall j < \text{length ts}. i\neq j \rightarrow 
\text{(let (-,-,sb,j,-,-) = ts!j in } \\
a \notin \text{all-acquired (takeWhile (Not o is-volatile-Write sb) sbj))} 
assumes a-ro-share: a \in \text{read-only (share sb S)} 
shows a \in \text{read-only (share (dropWhile (Not o is-volatile-Write sb) sb) (share-all-until-volatile-write ts S))}
\langle proof \rangle

lemma all-acquired-dropWhile-in: x \in \text{all-acquired (dropWhile P sb)} \implies x \in \text{all-acquired sb}
\langle proof \rangle
lemma all-acquired-takeWhile-in: \( x \in \text{all-acquired} \) (takeWhile P sb) \( \implies x \in \text{all-acquired} \) sb
(\( \text{proof} \))

lemmas all-acquired-takeWhile-dropWhile-in = all-acquired-takeWhile-in all-acquired-dropWhile-in

lemma split-in-read-only-reads:
\( \forall O. \ a \in \text{read-only-reads} \ O \ xs \implies (\exists t \ v \ ys \ zs. \ xs=ys @ \text{Read}_sb \ False \ a \ t \ v \ # \ zs \ \& \ a \notin \text{acquired} \ True \ ys \ O) \)
(\( \text{proof} \))

lemma insert-monoD: \( W \subseteq W' \implies \text{insert} \ a \ W \subseteq \text{insert} \ a \ W' \)
(\( \text{proof} \))

primrec unforwarded-non-volatile-reads:: \( 'a \ \text{memref} \ \text{list} \Rightarrow \text{addr set} \Rightarrow \text{addr set} \)
where
unforwarded-non-volatile-reads [] W = {} |
unforwarded-non-volatile-reads (x#xs) W =
(case x of
  Read \sb \ volatile \ a - - \Rightarrow (if a \notin W \land \neg \text{volatile} 
    then \text{insert} \ a (\text{unforwarded-non-volatile-reads} \ xs \ W) 
    else (\text{unforwarded-non-volatile-reads} \ xs \ W))
| \text{Write}_sb - a - - - - - \Rightarrow \text{unforwarded-non-volatile-reads} \ xs (\text{insert} \ a \ W) 
| - \Rightarrow \text{unforwarded-non-volatile-reads} \ xs \ W)

lemma unforwarded-non-volatile-reads-non-volatile-Read\sb:
\( \forall W. \ \text{unforwarded-non-volatile-reads} \ sb \ W \subseteq \text{outstanding-ref} \ \text{is-non-volatile-Read}_sb \ sb \)
(\( \text{proof} \))

lemma in-unforwarded-non-volatile-reads-non-volatile-Read\sb:
\( a \in \text{unforwarded-non-volatile-reads} \ sb \ W \implies a \in \text{outstanding-ref} \ \text{is-non-volatile-Read}_sb \ sb \)
(\( \text{proof} \))

lemma unforwarded-non-volatile-reads-antimono:
\( \forall W \ W'. \ W \subseteq W' \implies \text{unforwarded-non-volatile-reads} \ xs \ W' \subseteq \text{unforwarded-non-volatile-reads} \ xs \ W \)
(\( \text{proof} \))

lemma unforwarded-non-volatile-reads-antimono-in:
\( x \in \text{unforwarded-non-volatile-reads} \ xs \ W' \implies W \subseteq W' \implies x \in \text{unforwarded-non-volatile-reads} \ xs \ W \)
(proof)

**lemma** unforwarded-non-volatile-reads-append: \( \bigwedge W. \) unforwarded-non-volatile-reads (xs@ys) W =
(unforwarded-non-volatile-reads xs W \cup
unforwarded-non-volatile-reads ys (W \cup \text{outstanding-refs is-Write}_{sb} xs))

(proof)

**lemma** reads-consistent-mem-eq-on-unforwarded-non-volatile-reads:
assumes mem-eq: \( \forall a \in A \cup W. \ m' a = m a \)
assumes subset: unforwarded-non-volatile-reads sb W \subseteq A
assumes consis-m: reads-consistent pending-write \( O \) m sb
shows reads-consistent pending-write \( O \) m' sb

(proof)

**lemma** reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop:
assumes mem-eq: \( \forall a \in A \cup W. \ m' a = m a \)
assumes subset: unforwarded-non-volatile-reads (dropWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) W \subseteq A
assumes subset-acq: acquired-reads True (takeWhile (Not \circ \text{is-volatile-Write}_{sb}) sb) \( O \) \subseteq A
assumes consis-m: reads-consistent False \( O \) m sb
shows reads-consistent False \( O \) m' sb

(proof)

**lemma** read-only-read-witness:\( \bigwedge S \ O. \)
[non-volatile-owned-or-read-only True \( S \ O \) sb; \( a \in \text{read-only-reads} \ O \) sb]
\( \Rightarrow \)
\( \exists xs \ y s t \ v. \ sb=xs@ \text{Read}_{sb} \) False a t v \# ys \( \land a \in \text{read-only} \) (share xs \( S \)) \( \land a \notin \) read-only-reads \( \ O \) xs

(proof)

**lemma** read-only-read-acquired-witness: \( \bigwedge S \ O. \)
[non-volatile-owned-or-read-only True \( S \ O \) sb; sharing-consistent \( S \ O \) sb; \( a \notin \) read-only \( S \); a \( \notin \) \( O \); \( a \in \text{read-only-reads} \ O \) sb]
\( \Rightarrow \)
\( \exists xs \ y s t \ v. \ sb=xs@ \text{Read}_{sb} \) False a t v \# ys \( \land a \in \text{all-acquired} \) xs \( \land a \in \text{read-only} \) (share xs \( S \)) \( \land \)
\( a \notin \text{read-only-reads} \ O \) xs

(proof)
lemma unforwarded-not-written: \( \forall W. a \in \text{unforwarded-non-volatile-reads } sb \ W \implies a \notin W \)
\( \langle proof \rangle \)

lemma unforwarded-witness: \( \forall X. [a \in \text{unforwarded-non-volatile-reads } sb \ X] \implies \exists xs \ ys \ t \ v. sb=xs@ Read_{sb} False a t v \# ys \land a \notin \text{outstanding-refs } is-Write_{sb} xs \)
\( \langle proof \rangle \)

lemma read-only-read-acquired-unforwarded-witness: \( \forall S \ O. [\text{non-volatile-owned-or-read-only } True \ S \ O \ sb; \text{sharing-consistent } S \ O \ sb; a \notin \text{read-only } S; a \notin O; a \in \text{read-only-reads } O \ sb; a \in \text{unforwarded-non-volatile-reads } sb \ X ] \implies \exists xs \ ys \ t \ v. sb=xs@ Read_{sb} False a t v \# ys \land a \in \text{all-acquired } xs \land a \notin \text{outstanding-refs } is-Write_{sb} xs \)
\( \langle proof \rangle \)

lemma takeWhile-prefix: \( \exists ys. \text{takeWhile } P \ xs @ ys = xs \)
\( \langle proof \rangle \)

lemma unforwarded-empty-extend:
\[ \forall W. x \in \text{unforwarded-non-volatile-reads } sb \ \{\} \implies x \notin W \implies x \in \text{unforwarded-non-volatile-reads } sb \ W \]
\( \langle proof \rangle \)

lemma notin-unforwarded-empty:
\[ \forall W. a \notin \text{unforwarded-non-volatile-reads } sb \ W \implies a \notin W \implies a \notin \text{unforwarded-non-volatile-reads } sb \ \{\} \]
\( \langle proof \rangle \)

lemma
\begin{align*}
\text{assumes } ro: & a \in \text{read-only } S \implies a \in \text{read-only } S' \\
\text{assumes } & a\text{-in: } a \in \text{read-only } (S \oplus W \ R) \\
\text{shows } & a \in \text{read-only } (S' \oplus W \ R)
\end{align*}
\( \langle proof \rangle \)

lemma
\begin{align*}
\text{assumes } ro: & a \in \text{read-only } S \implies a \in \text{read-only } S' \\
\text{assumes } & a\text{-in: } a \in \text{read-only } (S \ominus A \ L) \\
\text{shows } & a \in \text{read-only } (S' \ominus A \ L)
\end{align*}
\( \langle proof \rangle \)

lemma non-volatile-owned-or-read-only-read-only-reads-eq:
\( \forall S \ S' \ O \ pending-write. \)
lemma non-volatile-owned-or-read-only-read-only-reads-eq':
\[ S' \]
\[ \forall a \in \text{read-only-reads } O \]
\[ a \in \text{read-only } S \rightarrow a \in \text{read-only } S' \]
\[ \Rightarrow \text{non-volatile-owned-or-read-only pending-write } S' O \]
\[ \langle \text{proof} \rangle \]

lemma no-write-to-read-only-memory-read-only-reads-eq:
\[ S' \]
\[ \forall a \in \text{read-only-reads } (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}) sb) O)
\]
\[ (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}) sb). a \in \text{read-only } S \rightarrow a \in \text{read-only } S' \]
\[ \Rightarrow \text{non-volatile-owned-or-read-only False } S' O \]
\[ \langle \text{proof} \rangle \]

lemma reads-consistent-drop:
\[ \text{reads-consistent False } O \]
\[ \Rightarrow \text{reads-consistent True }
\]
\[ (\text{acquired True } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}) sb) O)
\]
\[ (\text{flush } (\text{takeWhile } (\text{Not } \circ \text{is-volatile-Write}) sb) m)
\]
\[ (\text{dropWhile } (\text{Not } \circ \text{is-volatile-Write}) sb)
\]
\[ \langle \text{proof} \rangle \]

end

theory ReduceStoreBufferSimulation
imports ReduceStoreBuffer

begin

locale initial = simple-ownership-distinct + read-only-unowned + unowned-shared +
constrains ts::{p, p store-buffer, bool.owns.rels} thread-config list
assumes empty-sb: [i < length ts; ts!={p, is, xs, sb, D, O, R}] \[\Rightarrow\] sb=[]
assumes empty-is: [i < length ts; ts!={p, is, xs, sb, D, O, R}] \[\Rightarrow\] is=[]
assumes empty-rels: [i < length ts; ts!={p, is, xs, sb, D, O, R}] \[\Rightarrow\] R=Map.empty

sublocale initial \[\subseteq\] outstanding-non-volatile-refs-owned-or-read-only
\langle proof \rangle

sublocale initial \[\subseteq\] outstanding-volatile-writes-unowned-by-others
\langle proof \rangle

sublocale initial \[\subseteq\] read-only-reads-unowned
\langle proof \rangle

sublocale initial \[\subseteq\] ownership-distinct
\langle proof \rangle

sublocale initial \[\subseteq\] valid-ownership (proof)

sublocale initial \[\subseteq\] outstanding-non-volatile-writes-unshared
\langle proof \rangle

sublocale initial \[\subseteq\] sharing-consis
\langle proof \rangle

sublocale initial \[\subseteq\] no-outstanding-write-to-read-only-memory
\langle proof \rangle

sublocale initial \[\subseteq\] valid-sharing (proof)
sublocale initial \[\subseteq\] valid-ownership-and-sharing (proof)

sublocale initial \[\subseteq\] load-tmps-distinct
\langle proof \rangle

sublocale initial \[\subseteq\] read-tmps-distinct
\langle proof \rangle

sublocale initial \[\subseteq\] load-tmps-read-tmps-distinct
\langle proof \rangle

sublocale initial \[\subseteq\] load-tmps-read-tmps-distinct (proof)

sublocale initial \[\subseteq\] valid-write-sops
\langle proof \rangle

sublocale initial \[\subseteq\] valid-store-sops
\langle proof \rangle

sublocale initial \[\subseteq\] valid-sops (proof)

sublocale initial \[\subseteq\] valid-reads
\langle proof \rangle
sublocale initial_{sb} ⊆ valid-history
(proof)

sublocale initial_{sb} ⊆ valid-data-dependency
(proof)

sublocale initial_{sb} ⊆ load-tmps-fresh
(proof)

sublocale initial_{sb} ⊆ enough-flushs
(proof)

sublocale initial_{sb} ⊆ valid-program-history
(proof)

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 ts_{sb} m_{sb};
 S = share-all-until-volatile-write ts_{sb} S_{sb};
 length ts_{sb} = length ts;
 ∀ i < length ts_{sb}.
in
 let (p, is_{sb}, θ, sb, D_{sb}, O, R) = ts_{sb}!i;
suspends = dropWhile (Not ◦ is-volatile-Write_{sb}) sb
 in ∃ is D. instrs suspendsb ⊗ is_{sb} = is prog-instrs suspendsb ∧
 D_{sb} = (D ⊔ outstanding-refs is-volatile-Write_{sb} sb ≠ ∅) ∧
ts!i = (hd-prog p suspends,
 θ | (dom θ − read-tmps suspendsb),(),
 D,
 acquired True (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) O,
 release (takeWhile (Not ◦ is-volatile-Write_{sb}) sb) (dom S_{sb}) R )

]}
⇒
(ts_{sb},m_{sb},S_{sb}) ~ (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
(- ~ - [60,60] 100)

where
[length ts = length ts_{h};
 ∀ i < length ts.
 (∃ O’ D’ R’.
   let (p,is, θ, sb,D, O,R) = ts_{h}!i in
   ts!i=(p,is, θ, filter is-Write_{sb} sb,D’,O’,R’) ∧
 (filter is-Write_{sb} sb = [] → sb=[]))

]}
⇒
 ts ~_{h} ts_{h}

lemma (in initial_{sb}) history-refl:ts ~_{h} ts
(proof)

lemma share-all-empty: ∀ i p is xs sb D O R. i < length ts → ts!i=(p,is,xs,sb,D,O,R) → sb=[]
⇒ share-all-untill-volatile-write ts S = S
(proof)

121
lemma flush-all-empty: \( \forall i \ p \ is \ xs \ sb \ \mathcal{D} \ \mathcal{O} \ \mathcal{R}. \ i < length \ ts \implies ts!!=(p,is,xs,\mathcal{D},\mathcal{O},\mathcal{R}) \implies sb=[] \) 
\( \implies \text{flush-all-until-volatile-write} \ ts \ m = m \) 
\( \langle \text{proof} \rangle \)

lemma sim-config-emptyE:
assumes empty:
\( \forall i \ p \ is \ xs \ sb \ \mathcal{D} \ \mathcal{O} \ \mathcal{R}, \ i < length \ ts_{ab} \implies ts_{ab}!!=(p,is,xs,\mathcal{D},\mathcal{O},\mathcal{R}) \implies sb=[] \)
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 \( (p, is, \emptyset, sb, \mathcal{D}, \mathcal{O}, \mathcal{R}) = ts_{ab}!! \)
in \( ts!! = (p, is, \emptyset, (), \mathcal{D}, \mathcal{O}, \mathcal{R}) \))
\( \langle \text{proof} \rangle \)

lemma sim-config-emptyI:
assumes empty:
\( \forall i \ p \ is \ xs \ sb \ \mathcal{D} \ \mathcal{O} \ \mathcal{R}, \ i < length \ ts_{ab} \implies ts_{ab}!!=(p,is,xs,\mathcal{D},\mathcal{O},\mathcal{R}) \implies sb=[] \)
assumes leq: length \ ts = length \ ts_{ab}
assumes ts: \( (\forall i < length \ ts_{ab}. \)
let \( (p, is, \emptyset, sb, \mathcal{D}, \mathcal{O}, \mathcal{R}) = ts_{ab}!! \)
in \( ts!! = (p, is, \emptyset, (), \mathcal{D}, \mathcal{O}, \mathcal{R}) \))
shows \( (ts_{ab},m_{ab},S_{ab}) \sim (ts,m_{ab},S_{ab}) \)
\( \langle \text{proof} \rangle \)

lemma mem-eq-un-eq: \[ length \ ts' = length \ ts; \forall i < length \ ts'. P \ (ts''[i] = Q \ (ts''[i]) \] \implies (\( \bigcup x \in \text{set'}. P \ x \) = 
(\( \bigcup x \in \text{set}. Q \ x \))
\( \langle \text{proof} \rangle \)

lemma (in program) trace-to-steps:
assumes trace: trace \ c \ 0 \ k
shows steps: \( c \ 0 \implies c^* \ k \)
\( \langle \text{proof} \rangle \)

lemma (in program) safe-reach-to-safe-reach-upto:
assumes safe-reach: safe-reach-direct safe \( c_0 \)
shows safe-reach-upto n safe \( c_0 \)
\( \langle \text{proof} \rangle \)

lemma (in program-progress) safe-free-flowing-implies-safe-delayed':
assumes init: initial_{ab} \( ts_{ab}, S_{ab} \)
assumes sim: \( (ts_{ab},m_{ab},S_{ab}) \sim (ts,m,S) \)
assumes safe-free-ff: safe-reach-direct safe-free-flowing \( (ts,m,S) \)
shows safe-reach-direct safe-delayed \( (ts,m,S) \)
\( \langle \text{proof} \rangle \)

lemma map-ons-sb-owned: \( \forall j. \ j < length \ ts \implies \text{map} \ \mathcal{O}-\text{sb} \ ts \ ! j = (\mathcal{O}, \text{sb}_j) \implies \text{map owned} \ ts \ ! j = \mathcal{O}_j \)
\( \langle \text{proof} \rangle \)

lemma map-ons-sb-owned': \( \forall j. \ j < length \ ts \implies \mathcal{O}-\text{sb} \ (ts \ ! j) = (\mathcal{O}, \text{sb}_j) \implies \text{owned} \ (ts \ ! j) = \mathcal{O}_j \)
\( \langle \text{proof} \rangle \)

lemma read-only-read-acquired-unforwarded-acquire-witness:
\( \forall S \ O \ X. [\text{non-volatile-owned-or-read-only True} \ S \ O \ sb; \)
sharing-consistent \( S \ O \ sb; \ a \notin \text{read-only} \ S; a \notin \mathcal{O}; \)
a \in \text{unforwarded-non-volatile-reads sb X}\]
\( \implies (\exists \text{sop a' v ys zs A L R W} \)
\( \text{sb} = \text{ys} \ @ \ Write_{ab} \text{ True a' sop v A L R W} \neq \text{zs} \)
\( a \in \mathcal{A} \land a \notin \text{outstanding-refs is-Write}_{ab} \text{ ys } \land \ a' \neq a \ ) \lor \)
122
∃ A L R W ys zs. sb = ys ⊖ ① Ghost_w A L R W# zs ∧ a ∈ A ∧ a /∈ outstanding-refs is-Write_w ys)
⟨proof⟩

lemma release-shared-exchange-weak:
asumes shared-eq: ∀ a ∈ O ∪ all-acquired sb. (S\':shared) a = S a
asumes consis: weak-sharing-consistent O sb
shows release sb (dom S\') R = release sb (dom S) R
⟨proof⟩

lemma read-only-share-all-shared: \( \forall S. [ a \in \text{read-only (share sb S)}] \)
implies a ∈ read-only S ∪ all-shared sb
⟨proof⟩

lemma read-only-shared-all-until-volatile-write-subset:\( \forall S. \)
read-only (share-all-until-volatile-write ts S) ⊆
read-only S ∪ (\( \bigcup ((\lambda (-,-,-,sb,-,-). \text{all-shared (takeWhile (Not o is-volatile-Write_w sb))} ) \text{'} set ts)) \)
⟨proof⟩

lemma read-only-share-acquired-all-shared:
\( \forall S. \text{weak-sharing-consistent O sb} \implies O \cap \text{read-only S} = \{\} \implies \)
a ∈ read-only (share sb S) ⇒ a ∈ O ∪ all-acquired sb ⇒ a ∈ all-shared sb
⟨proof⟩

lemma read-only-share-unowned:\( \forall S. \)
[weak-sharing-consistent O sb; O ∩ read-only S = \{\}; a /∈ O ∪ all-acquired sb; a ∈ read-only S]
⇒ a ∈ read-only (share sb S)
⟨proof⟩

lemma release-False-mono:
\( \forall S R. R a = \text{Some False} \implies \text{outstanding-refs is-volatile-Write_w sb} = \{\} \implies \)
release sb S R a = Some False
⟨proof⟩

lemma release-False-mono-take:
\( \forall S R. R a = \text{Some False} \implies \text{release (takeWhile (Not o is-volatile-Write_w sb) S R a} = \text{Some False} \)
⟨proof⟩

lemma shared-switch:
\( \forall S O. [\text{weak-sharing-consistent O sb}; \text{read-only S} \cap O = \{\}; \)
S a ≠ Some False; share sb S a = Some False]
⇒ a ∈ O ∪ all-acquired sb
⟨proof⟩

lemma shared-switch-release-False:
\( \forall S R. [\)
outstanding-refs is-volatile-Write_w sb = \{\};
a /∈ dom S;
a ∈ dom (share sb S)]
⇒
release sb (dom S) R a = Some False
⟨proof⟩
lemma release-not-unshared-no-write:
\[ S R. \]
outstanding-refs is-volatile-Write\(_{ab}\) sb = \{\};
non-volatile-writes-unshared \( S \) sb;
release sb (dom \( S \)) \( R \) a ≠ Some False;
a ∈ dom (share sb \( S \))]
⇒
a \∉ outstanding-refs is-non-volatile-Write\(_{ab}\) sb
<proof/>
corollary release-not-unshared-no-write-take:
assumes nww: non-volatile-writes-unshared \( S \) (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb)
assumes rel: release (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb) (dom \( S \)) \( R \) a ≠ Some False
assumes a-in: a ∈ dom (share (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb)) \( S \)
shows
a \∉ outstanding-refs is-non-volatile-Write\(_{ab}\) (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb)
<proof/>
lemma read-only-unacquired-share:
\[ S O. \]
\[ O \cap \text{read-only} S = \{\}; \text{weak-sharing-consistent} O sb; a ∈ \text{read-only} S; \]
a \∉ all-shared sb; a \∉ acquired True sb \( O \) \]
⇒ a ∈ read-only (share sb \( S \))
<proof/>
lemma read-only-share-all-until-volatile-write-unacquired:
\[ S. \]
ownership-distinct ts; read-only-unknown \( S \) ts; weak-sharing-consis ts;
\( ∀ i < \text{length} ts. (\text{let} (\ldots, sb, \ldots, O, R) = ts! i \text{ in} \)
a \∉ acquired True (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb) \( O \) \∧
a \∉ all-shared (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb)
)
⇒ a ∈ read-only \( S \)
⇒ a ∈ read-only (share-all-until-volatile-write ts \( S \))
<proof/>
lemma not-shared-not-acquired-switch:
\[ X Y. \]
a \∉ all-shared sb; a \∉ X; a \∉ acquired True sb X; a \∉ Y \⇒ a \∉ acquired True sb Y
<proof/>
lemma read-only-share-all-acquired-in:
\[ S O. \]
\( O \cap \text{read-only} S = \{\}; \text{weak-sharing-consistent} O sb; a \in \text{read-only} (\text{share sb} S) \]
⇒ a ∈ read-only (share sb Map.empty) \( X \) (a ∈ read-only \( S \) \( \land \) a \∉ acquired True sb \( O \) \( \land \) a \∉ all-shared sb)
<proof/>
lemma in-read-only-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, is, \theta, sb, D, O, R)
assumes a-unacquired-others: \( ∀ j < \text{length} ts. i \neq j \rightarrow \)
(\text{let} (\ldots, sb, \ldots, O, \ldots) = ts! j \text{ in} \)
a \∉ acquired True (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb) \( O \) \∧
a \∉ all-shared (takeWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb) \( j \))
assumes a-ro-share: a ∈ read-only (share sb \( S \))
shows a ∈ read-only (share (dropWhile (Not \( o \) is-volatile-Write\(_{ab}\)) sb)}\]
lemma all-acquired-unshared-acquired:
\[ \forall O. \ a \in \text{all-acquired sb} \implies a \notin \text{all-shared sb} \implies a \in \text{acquired True sb} \]

lemma safe-RMW-common:
assumes safe: \( O \subseteq R \subseteq S \)
shows \( (\forall a \in O \setminus a \in \text{dom } S) \land (\forall j < \text{length } O. i \neq j \rightarrow (R \setminus j) a \neq \text{Some False}) \)

lemma acquired-reads-all-acquired':
\[ \forall O. \ \text{acquired-reads True sb } O \subseteq \text{acquired True sb } O \cup \text{all-shared sb} \]

lemma release-all-shared-exchange:
\[ \forall R \subseteq S. \forall a \in \text{all-shared sb}. \ (a \in S') = (a \in S) \implies \text{release sb } S' \ R = \text{release sb } S \ R \]

lemma release-append-Prog:
\[ \forall S \ R. \ (\text{release } (\text{takeWhile } \neg \text{is-volatile-Write sb}) \ (\text{sb } \ @ \ [\text{Prog sb } p_1 p_2 \ mis])) \ S \ R) = \]
\[ (\text{release } (\text{takeWhile } \neg \text{is-volatile-Write sb}) \ (\text{sb } \ @ \ [\text{Prog sb } p_1 p_2 \ mis])) \ S \ R) \]

A.5 Simulation of Store Buffer Machine with History by Virtual Machine with Delayed Releases

theorem (in xvalid-program) concurrent-direct-steps-simulates-store-buffer-history-step:
assumes step-sb: \((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbh} (ts_{sb}', m_{sb}', S_{sb}')\)
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 tmpp-distinct: tmpp-distinct ts_{sb}
assumes valid-sops: valid-sops ts_{sb}
assumes valid-dd: valid-data-dependency ts_{sb}
assumes load-tmpp-fresh: load-tmpp-fresh ts_{sb}
assumes enough-flushs: enough-flushs ts_{sb}
assumes valid-program-history: valid-program-history ts_{sb}
assumes valid: valid ts_{sb}
assumes sim: \((ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)\)
assumes safe-reach: safe-reach-direct safe-delayed \((ts, m, S)\)
shows valid-ownership \( S_{sb}' ts_{sb}' \land \text{valid-reads } m_{sb}' ts_{sb}' \land \text{valid-history program-step } ts_{sb}' \land \]
\[ \text{valid-sharing } S_{sb}' ts_{sb}' \land \text{tmpp-distinct } ts_{sb}' \land \text{valid-data-dependency } ts_{sb}' \land \]
\[ \text{valid-sops } ts_{sb}' \land \text{load-tmpp-fresh } ts_{sb}' \land \text{enough-flushs } ts_{sb}' \land \]
\[ \text{valid-program-history } ts_{sb}' \land \text{valid } ts_{sb}' \land \]
\[ (\exists ts' S' m'. (ts, m, S) \Rightarrow^*_d (ts', m', S') \land \]
\[ (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S')) \]
\begin{proof}

\textbf{theorem (in xvalid-program)} concurrent-direct-steps-simulates-store-buffer-history-steps:
\begin{itemize}
  \item \textbf{assumes} step-sb: \((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow ts_{sb}'\) \((ts_{sb}', m_{sb}', S_{sb}')\)
  \item \textbf{assumes} valid-own: valid-ownership \(S_{sb} ts_{sb}\)
  \item \textbf{assumes} valid-sb-reads: valid-reads \(m_{sb} ts_{sb}\)
  \item \textbf{assumes} tmpls-distinct: \(tmps\)-distinct \(ts_{sb}\)
  \item \textbf{assumes} valid-sops: valid-sops \(ts_{sb}\)
  \item \textbf{assumes} valid-dd: valid-data-dependency \(ts_{sb}\)
  \item \textbf{assumes} load-tmpls-fresh: \(load\)-tmpl\(s\)-fresh \(ts_{sb}\)
  \item \textbf{assumes} enough-flushs: enough-flushs \(ts_{sb}\)
  \item \textbf{assumes} valid-program-history: valid-program-history \(ts_{sb}\)
  \item \textbf{assumes} valid: valid \(ts_{sb}\)
\end{itemize}
\textbf{shows} \(\\land ts \ S \ m. \ (ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S) \Rightarrow \text{safe-reach-direct safe-delayed} \ (ts, m, S) \Rightarrow \)
\begin{itemize}
  \item valid-ownership \(S_{sb}' ts_{sb}' \land valid-reads m_{sb}' ts_{sb}' \land valid-history program-step ts_{sb}'\)
  \item valid-sharing \(S_{sb}' ts_{sb}' \land \text{tmpls-distinct} ts_{sb}' \land \text{valid-data-dependency} ts_{sb}' \land \)
  \item valid-sops \(ts_{sb}' \land load\)-tmpl\(s\)-fresh \(ts_{sb}' \land \text{enough-flushs} ts_{sb}' \land \)
  \item valid-program-history \(ts_{sb}' \land valid ts_{sb}'\)
\end{itemize}
\(\exists ts' m' S'. \ (ts, m, S) \Rightarrow d^* (ts', m', S') \land (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S')\)
\end{proof}

\textbf{sublocale} initial_{sb} \subseteq tmpls-distinct \(\langle proof \rangle\)

\textbf{locale} xvalid-program-progress = program-progress + xvalid-program

\textbf{theorem (in xvalid-program-progress)} concurrent-direct-execution-simulates-store-buffer-history-execution:
\begin{itemize}
  \item \textbf{assumes} exec-sb: \((ts_{sb}, m_{sb}, S_{sb}) \Rightarrow ts_{sb}'\) \((ts_{sb}', m_{sb}', S_{sb}')\)
  \item \textbf{assumes} init: initial_{sb} \(ts_{sb} S_{sb}\)
  \item \textbf{assumes} valid: valid \(ts_{sb}\)
  \item \textbf{assumes} sim: \((ts_{sb}, m_{sb}, S_{sb}) \sim (ts, m, S)\)
  \item \textbf{assumes} safe: safe-reach-direct safe-free-flowing \((ts, m, S)\)
\end{itemize}
\textbf{shows} \(\exists ts' m' S'. \ (ts, m, S) \Rightarrow d^* (ts', m', S') \land\)
\((ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S')\)
\end{proof}

\textbf{lemma} filter-is-Write_{sb}-Cons-Write_{sb}: filter is-Write_{sb} \(xs = \text{Write}_{sb}\) volatile a sop \(v\ A\ L\ R\ W\ R\ W\ #ys\)
\begin{itemize}
  \item \textbf{shows} \(\exists rs\ rws. \ (\forall r \in \text{set} \ r. \ is\)-Read_{sb} \(r\ \lor\ is\)-Prog_{sb} \(r\ \lor\ is\)-Ghost_{sb} \(r\)\) \land \(xs = rs@\text{Write}_{sb}\) volatile a sop \(v\ A\ L\ R\ W\ #rws\ \land\ ys = \text{filter is-Write}_{sb}\) rws
\end{itemize}
\begin{proof}

\end{proof}

126
lemma filter-is-Write_sb-empty: filter is-Write sb xs = []
  \implies (\forall r \in \text{set } xs. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r)
  
  ⟨proof⟩

lemma flush-reads-program: \forall O S R.
  \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r \implies 
  \exists O' R' S'. (m,sb,O,R,S) \rightarrow^* (m,\epsilon,O',R',S')
  
  ⟨proof⟩

lemma flush-progress: \exists m'. O' R' S'. (m,r#sb,O,R,S) \rightarrow_f (m',sb,O',R',S')
  
  ⟨proof⟩

lemma flush-empty: \exists sb'. \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r \implies 
  \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r \implies 
  \exists sb'. \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r
  
  ⟨proof⟩

lemma flush-append: \exists m'. O' R' S'. (m,r#sb,O,R,S) \rightarrow_f (m',sb,O',R',S')
  
  ⟨proof⟩

lemmas store-buffer-step-induct =
  store-buffer-step.induct [split-format (complete), consumes 1, case-names SBWrite sb]

theorem flush-simulates-filter-writes:
  \exists sb'. \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r \implies 
  \forall m', sb', O', R', S'. (m,sb,O,R,S) \rightarrow^* (m',sb,O',R',S') \land 
  \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r \implies 
  \forall m', sb', O', R', S'. (m,sb,O,R,S) \rightarrow^* (m',sb',O',R',S') \land 
  \forall r \in \text{set sb}. \text{is-Read sb } r \lor \text{is-Prog sb } r \lor \text{is-Ghost sb } r
  
  ⟨proof⟩

lemma buffered-val-filter-is-Write_sb-eq-ext:
  buffered-val (filter is-Write sb a) = buffered-val sb a
  
  ⟨proof⟩

lemma buffered-val-filter-is-Write_sb-eq:
  buffered-val (filter is-Write sb a) = buffered-val sb
  
  ⟨proof⟩

lemma outstanding-refs-is-volatile-Write_sb-filter-writes:
  outstanding-refs is-volatile-Write sb xs = 
  outstanding-refs is-volatile-Write sb xs
  
  ⟨proof⟩

127
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) →_{sb} (ts', m', S')  
assumes sim: ts ∼_h ts_h  
shows ∃ts_h', S_h': (ts_h, m, S_h) ⇒_{sb} (ts_h', m', S_h') ∧ ts' ∼_h ts_h'  
⟨proof⟩

**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} (ts_h', m', S_h') ∧ ts' ∼_h ts_h'  
⟨proof⟩

**theorem (in valid-program-progress) concurrent-direct-execution-simulates-store-buffer-execution:**  
assumes exec-sb: (ts_{sb}, m_{sb}, x) →_{sb} (ts_{sb}', m_{sb}', x')  
assumes init: initial_{sb} ts_{sb} S_{sb}  
assumes valid: valid ts_{sb}  
assumes sim: (ts_{sb}, m_{sb}, S_{sb}) ∼ (ts, m, S)  
shows ∃ts_h', S_h', ts' m' S'.  
(ts_{sb}, m_{sb}, S_{sb}) →_{sb} (ts_h', m_{sb}', S_h') ∧  
ts_{sb} ∼_h ts_h' ∧  
(ts, m, S) ⇒_{d} (ts', m', S') ∧  
(ts_h', m_{sb}', S_h') ∼ (ts', m', S')  
⟨proof⟩

**inductive sim-direct-config:**  
('p', p store-buffer, 'dirty', 'owns', 'rels) thread-config list ⇒ ('p, unit, bool, 'owns', 'rels') thread-config list ⇒ bool  
(- ~_d - [60, 60] 100)  
where  
|length ts = length ts_d;  
∀i < length ts.  
(∃O' D' R').  
let (p, is, â, sb, D, O, R) = ts_d!i in  
  ts!i = (p, is, â, [], 'D', O', R')  
|  
⇒  
ts ∼_d ts_d

**lemma empty-sb-sims:**  
assumes empty:  
∀i p is xs sb D O R. i < length ts_{sb} → ts_{sb}!i = (p, is, xs, sb, D, O, R) → sb=[]  
assumes sim-h: ts_{sb} ∼_h ts_h  
assumes sim-d: (ts_h, m_h, S_h) ∼ (ts, m, S)
shows \( ts_{sb} \sim_d ts \land m_h = m \land \text{length } ts_{sb} = \text{length } ts \)

\[\langle \text{proof} \rangle\]

**Lemma** empty-d-sims:

**Assumes** sim: \( ts_{sb} \sim_d ts \)

**Shows** \( \exists tsh. \, ts_{sb} \sim_h tsh \land (tsh, m, S) \sim (ts, m, S) \)

\[\langle \text{proof} \rangle\]

**Theorem** (in xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-execution-empty:

**Assumes** exec-sb: \((ts_{sb}, m_{sb}, x) \Rightarrow^* (ts_{sb}', m_{sb}', x')\)

**Assumes** init: initial_{sb} ts_{sb} S_{sb}

**Assumes** valid: valid ts_{sb}

**Assumes** empty-rels: \([i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, R)] \Rightarrow R = \text{Map.empty}\)

locale initial_{d} = simple-ownership-distinct + read-only-unowned + unowned-shared + fixes valid

**Assumes** empty-is: \([i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, R)] \Rightarrow is = []\)

**Assumes** empty-rels: \([i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, R)] \Rightarrow R = \text{Map.empty}\)

**Assumes** valid-init: valid \((\lambda (p, is, \emptyset, sb, D, O, R). \ (p, is, \emptyset, [], D, O, R))\) ts)

locale empty-store-buffers =

**Fixes** ts::\(\langle p, p \text{ thread-config bo}u\text{ll, owns, rels} \rangle\) thread-config list

**Assumes** empty-sb: \([i < \text{length } ts; ts!i = (p, is, xs, sb, D, O, R)] \Rightarrow sb = []\)

**Lemma** initial-d-sb:

**Assumes** init: initial_{d} ts S valid

**Shows** initial_{sb} \((\lambda (p, is, \emptyset, sb, D, O, R). \ (p, is, \emptyset, [], D, O, R))\) ts) S

\(\text{(is initial}_{sb} \ ? \text{map } S)\)

\[\langle \text{proof} \rangle\]

**Theorem** (in xvalid-program-progress) store-buffer-execution-result-sequential-consistent:

**Assumes** exec-sb: \((ts_{sb}, m, x) \Rightarrow^* (ts_{sb}', m', x')\)

**Assumes** empty': empty-store-buffers ts_{sb}'

**Assumes** sim: \( ts_{sb} \sim_d ts \)

**Assumes** init: initial_{d} ts S valid

**Assumes** safe: safe-reach-direct safe-free-flowing (ts, m, S)

**Shows** \( \exists ts' S'. \ (ts, m, S) \Rightarrow^d (ts', m', S') \land ts_{sb}' \sim_d ts' \)

\[\langle \text{proof} \rangle\]

locale initial_{p} = 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)] \implies is=\[]\)
assumes valid-init: valid (map (\(\lambda\((p, is, \emptyset, sb, D, O, R)\). \((p, is, \emptyset, [], D, O, \text{Map.} \text{empty})\)) ts)

\[\text{assumes exec-sb: } ts_{sb} \implies_{sb}^* (ts'_{sb}, m', x')\]
assumes empty': empty-store-buffers ts_{sb}'
assumes init: initial ts S valid
assumes safe: safe-reach-virtual-safe-free-flowing \((ts, m, S)\)

shows \(\exists ts' S'.\)
\((ts, m, S) \implies_v^* (ts', m', S') \land ts'_{sb} \sim_d ts'\)

\(\langle\text{proof}\rangle\)

A.7 Plug Together the Two Simulations

corollary (in xvalid-program) concurrent-direct-steps-simulates-store-buffer-step:
assumes step-sb: \((ts_{sb}, m_{sb}, S_{sb}) \implies_{sb} (ts'_{sb}, m'_{sb}, S'_{sb})\)
assumes sim-h: \(ts_{sbh} \sim_h ts_{sbh}\)
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-hist: 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 \(\exists ts_{sbh}', S_{sbh}'.\)
\((ts_{sbh}, m_{sb}, S_{sbh}) \implies_{sbh}^* (ts'_{sbh}, m'_{sb}, S'_{sbh}) \land ts'_{sbh} \sim_h ts'_{sbh} \land\)
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}' \land
(\(\exists ts' S' m'. (ts, m, S) \implies_d^* (ts', m', S') \land\)
\((ts'_{sbh}, m'_{sb}, S'_{sbh}) \sim (ts', m', S')\))

\(\langle\text{proof}\rangle\)

lemma conj-commI: \(P \land Q \implies Q \land P\)

\(\langle\text{proof}\rangle\)
lemma def-to-eq: \( P = Q \implies P \equiv Q \)

(proof)

context xvalid-program
begin

definition
invariant ts \( S \) m \( \equiv \)
valid-ownership \( S \) ts \& valid-reads m ts \& valid-history program-step ts \&
valid-sharing \( S \) ts \& tmtps-distinct ts \& valid-data-dependency ts \&
valid-sops ts \& load-tmps-fresh ts \& enough-flushs ts \& valid-program-history ts \&
valid ts

definition ownership-inv \( \equiv \) valid-ownership

definition temporaries-inv ts \( \equiv \) tmtps-distinct ts \& load-tmps-fresh ts

definition history-inv ts m \( \equiv \) valid-history program-step ts \& valid-program-history ts \&
valid-reads m ts

definition data-dependency-inv ts \( \equiv \) valid-data-dependency ts \& load-tmps-fresh ts \&
valid-sops ts

definition barrier-inv \( \equiv \) enough-flushs

lemma invariant-grouped-def: invariant ts \( S \) m \( \equiv \)
ownership-inv \( S \) ts \& sharing-inv \( S \) ts \& temporaries-inv ts \& data-dependency-inv ts \&
history-inv ts m \& barrier-inv ts \& valid ts

(proof)

theorem (in xvalid-program) simulation' :
assumes step-sb: \( (ts_{sb}, m_{sb}, S_{sb}) \Rightarrow_{sbb} (ts_{sb}', m_{sb}', S_{sb}') \)
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}' \&
\( \exists ts' m'. (ts, m, S) \Rightarrow^* (ts', m', S') \land (ts_{sb}', m_{sb}', S_{sb}') \sim (ts', m', S') \)

(proof)

lemmas (in xvalid-program) simulation = conj-commI [OF simulation']
end

end

A.8 PIMP

theory PIMP
imports ReduceStoreBufferSimulation
begin

datatype expr = Const val | Mem bool addr | Tmp sop
| Unop val \Rightarrow val expr

end

131
| Binop val ⇒ val ⇒ val expr expr

datatype stmt =
  Skip
  | Assign bool expr expr tmps ⇒ owns tmtps ⇒ owns tmmps ⇒ owns
owns
  | CAS expr expr expr tmps ⇒ owns tmmps ⇒ owns tmmps ⇒ owns
  | Seq stmt stmt
  | Cond expr stmt stmt
  | While expr stmt
  | SGhost tmmps ⇒ owns tmmps ⇒ owns tmmps ⇒ owns tmmps ⇒ 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
  — trick to enforce sop to be sensible in the current context, without
having to include wellformedness constraints
| eval-expr t (Unop f e) = (let (D, f_e) = eval-expr t e in (D, λθ. f (f_e θ))
| eval-expr t (Binop f e_1 e_2) = (let (D_1, f_1) = eval-expr t e_1;
 (D_2, f_2) = eval-expr (t + (used-tmps e_1)) e_2
 in (D_1 ∪ D_2, λθ. f (f_1 θ) (f_2 θ)))

primrec valid-sops-expr:: nat ⇒ expr ⇒ bool
where
valid-sops-expr t (Const v) = True
| valid-sops-expr t (Mem volatile a) = True
| valid-sops-expr t (Tmp sop) = ((∀ t' ∈ fst sop. t' < t) ∧ valid-sop sop)
|valid-sops-expr t (Unop f e) = valid-sops-expr t e
|valid-sops-expr t (Binop f e₁ e₂) = (valid-sops-expr t e₁ ∨ valid-sops-expr t e₂)

**primrec** valid-sops-stmt:: nat ⇒ stmt ⇒ bool
**where**
valid-sops-stmt t Skip = True
valid-sops-stmt t (Assign volatile a e A L R W) = (valid-sops-expr t a ∧ valid-sops-expr t e)
valid-sops-stmt t (CAS a c e s A L R W) = (valid-sops-expr t a ∧ valid-sops-expr t c ∧ valid-sops-expr t s)
valid-sops-stmt t (Seq s₁ s₂) = (valid-sops-stmt t s₁ ∧ valid-sops-stmt t s₂)
valid-sops-stmt t (Cond e s₁ s₂) = (valid-sops-expr t e ∧ valid-sops-stmt t s₁ ∧ valid-sops-stmt t s₂)
valid-sops-stmt t (While e s) = (valid-sops-expr t e ∧ valid-sops-stmt t s)
valid-sops-stmt t (SGhost A L R W) = True
valid-sops-stmt t SFence = True

type-synonym stmt-config = stmt × nat
**consts**
isTrue:: val ⇒ bool

**inductive** stmt-step:: tmps ⇒ stmt-config ⇒ stmt-config × instrs ⇒ bool
(\not⊢ \not→ s - [60,60,60] 100)
**for** \(\emptyset\)
**where**

| AssignAddr:
\(\forall\ sop.\ a \not\equiv Tmp\ sop \implies θ ⊢ (Assign\ volatile\ a\ e\ A\ L\ R\ W,\ t) \rightarrow s\)
\(((Assign\ volatile\ (Tmp\ (eval-expr\ t\ a))\ e\ A\ L\ R\ W,\ t + used-tmps\ a),\ issue-expr\ t\ a)\)

| Assign:
\(D \subseteq\ dom\ θ\ \implies θ ⊢ (Assign\ volatile\ (Tmp\ (D, a))\ e\ A\ L\ R\ W,\ t) \rightarrow s\)
\(((Skip,\ t + used-tmps\ e),\ issue-expr\ t\ e@\Write\ volatile\ (a\ θ)\ (eval-expr\ t\ e)\ (A\ θ)\ (L\ θ)\ (R\ θ)\ (W\ θ))\)

| CASAddr:
\(\forall\ sop.\ a \not\equiv Tmp\ sop \implies θ ⊢ (CAS\ a\ c_e\ s_e\ A\ L\ R\ W,\ t) \rightarrow s\)
\(((CAS\ (Tmp\ (eval-expr\ t\ a))\ c_e\ s_e\ A\ L\ R\ W,\ t + used-tmps\ a),\ issue-expr\ t\ a)\)

| CASComp:
\(\forall\ sop.\ c_e \not\equiv Tmp\ sop \implies θ ⊢ (CAS\ (Tmp\ (D_a, a))\ c_e\ s_e\ A\ L\ R\ W,\ t) \rightarrow s\)
\[
((\text{CAS (Tmp (D}_{a, a})) (\text{Tmp (eval-expr t c)})) s_e A L R W, t + \text{used-tmps } c_e), \\
\text{issue-expr t c_e})
\]

| CAS: |
\[
[D_a \subseteq \text{dom } \vartheta; D_c \subseteq \text{dom } \vartheta; \text{eval-expr t } s_e = (D, f)] \\
\implies \\
\vartheta \vdash (\text{CAS (Tmp (D}_{a, a})) (\text{Tmp (D}_{c, c})) s_e A L R W, t) \rightarrow_s \\
((\text{Skip, Suc (t + used-tmps s_e)}), \text{issue-expr t } s_e) \\
\text{RMW (a } \vartheta) (t + \text{used-tmps } s_e) (D, f) (\lambda \vartheta, \text{the (} \vartheta (t + \text{used-tmps } s_e) = c \vartheta) (\lambda v_1 v_2. v_1) \\
(A \vartheta) (L \vartheta) (R \vartheta) (W \vartheta) []
\]

| Seq: |
\[
\vartheta \vdash (s_1, t) \rightarrow_s ((s_1', t'), \text{is}) \\
\implies \\
\vartheta \vdash (\text{Seq } s_1 s_2, t) \rightarrow_s ((\text{Seq } s_1' s_2, t'), \text{is})
\]

| SeqSkip: |
\[
\vartheta \vdash (\text{Seq Skip } s_2, t) \rightarrow_s ((s_2, t), [])
\]

| Cond: |
\[
\forall \text{sop. } e \neq \text{Tmp sop} \\
\implies \\
\vartheta \vdash (\text{Cond e } s_1 s_2, t) \rightarrow_s \\
((\text{Cond (Tmp (eval-expr t e)) } s_1 s_2, t + \text{used-tmps } e), \text{issue-expr t } e)
\]

| CondTrue: |
\[
[D \subseteq \text{dom } \vartheta; \text{isTrue (} e \vartheta)]] \\
\implies \\
\vartheta \vdash (\text{Cond (Tmp (D}_{c, e})) s_1 s_2, t) \rightarrow_s ((s_1, t).[])
\]

| CondFalse: |
\[
[D \subseteq \text{dom } \vartheta; \neg \text{isTrue (} e \vartheta)]] \\
\implies \\
\vartheta \vdash (\text{Cond (Tmp (D}_{c, e})) s_1 s_2, t) \rightarrow_s ((s_2, t).[])
\]

| While: |
\[
\vartheta \vdash (\text{While } e s, t) \rightarrow_s \\
((\text{Cond e (Seq s (While e s)) Skip, t).[]})
\]

| SGhost: |
\[
\vartheta \vdash (\text{SGhost A L R W, t}) \rightarrow_s ((\text{Skip, t)}, [\text{Ghost (A } \vartheta) (L \vartheta) (R \vartheta) (W \vartheta)])
\]

| SFence: |
\[
\vartheta \vdash (\text{SFence, t}) \rightarrow_s ((\text{Skip, t)}, [\text{Fence}])
\]

**inductive-cases** stmt-step-cases [cases set]:
\[
\vartheta \vdash (\text{Skip, t}) \rightarrow_s c \\
\vartheta \vdash (\text{Assign volatile a e A L R W, t}) \rightarrow_s c
\]
\[\theta \vdash (\text{CAS } a \ c \ s \ w \ A \ L \ R \ W, \ t) \rightarrow s \ c\]
\[\theta \vdash (\text{Seq } s_1 \ s_2, \ t) \rightarrow s \ c\]
\[\theta \vdash (\text{Cond } e \ s_1 \ s_2, \ t) \rightarrow s \ c\]
\[\theta \vdash (\text{While } e \ s \ t) \rightarrow s \ c\]
\[\theta \vdash (\text{SGhost } A \ L \ R \ W, \ t) \rightarrow s \ c\]
\[\theta \vdash (\text{SFence }, \ t) \rightarrow s \ c\]

**Lemma valid-sops-expr-mono:** \[\forall t \ t'. \text{valid-sops-expr } t \ e \implies t \leq t' \implies \text{valid-sops-expr } t' \ e\]

**Lemma valid-sops-stmt-mono:** \[\forall t \ t'. \text{valid-sops-stmt } t \ s \implies t \leq t' \implies \text{valid-sops-stmt } t' \ s\]

**Lemma valid-sops-stmt-invariant:** \[\forall t. \text{valid-sops-stmt } t \ s \implies \text{valid-sops-stmt } t' \ s'\]

**Lemma stmt-step-tmps-count-mono:**

- **Assumes** step: \[\theta \vdash (s, t) \rightarrow s' ((s', t').is)\]
- **Shows** \[t \leq t'\]

**Lemma valid-sops-stmt-invariant:**

- **Assumes** step: \[\theta \vdash (s, t) \rightarrow s' ((s', t').is)\]
- **Shows** \[\text{valid-sops-stmt } t \ s \implies \text{valid-sops-stmt } t' \ s'\]

**Lemma map-le-restrict-map-eq:** \[m_1 \subseteq_m m_2 \implies D \subseteq \text{dom } m_1 \implies m_2 \upharpoonright D = m_1 \upharpoonright D\]

**Lemma sbh-step-preserves-load-tmps-bound:**

- **Assumes** step: \[(is, O, D, \emptyset, sb, S, m) \rightarrow_{sbh} (is', O', D', \emptyset', sb', S', m')\]
- **Assumes** less: \[\forall i \in \text{load-tmps } is. \ i < n\]
- **Shows** \[\forall i \in \text{load-tmps } is'. \ i < n\]

**Lemma sbh-step-preserves-read-tmps-bound:**

- **Assumes** step: \[(is, \emptyset, sb, m, D, O, S) \rightarrow_{sbh} (is', \emptyset', sb', m', D', O', S')\]
- **Assumes** less-is: \[\forall i \in \text{load-tmps } is. \ i < n\]
- **Assumes** less-sb: \[\forall i \in \text{read-tmps } sb. \ i < n\]
\( \forall i \in \text{read-tmps sb}'. i < n \)  
\(\langle\text{proof}\rangle\)

**lemma** sbh-step-preserves-tmps-bound:
- **assumes** step: \((\text{is},\theta,\text{sb},m,D,O,S) \rightarrow_{\text{sbh}} (\text{is}',\theta',\text{sb}',m',D',O',S')\)
- **assumes** less-dom: \(\forall i \in \text{dom } \theta. i < n\)
- **assumes** less-is: \(\forall i \in \text{load-tmps is}. i < n\)
- **shows** \(\forall i \in \text{dom } \theta'. i < n\)

\(\langle\text{proof}\rangle\)

**lemma** flush-step-preserves-read-tmps:
- **assumes** step: \((m,\text{sb},O) \rightarrow_{f} (m',\text{sb}',O')\)
- **assumes** less-sb: \(\forall i \in \text{read-tmps sb}. i < n\)
- **shows** \(\forall i \in \text{read-tmps sb}'. i < n\)

\(\langle\text{proof}\rangle\)

**lemma** flush-step-preserves-write-sops:
- **assumes** step: \((m,\text{sb},O) \rightarrow_{f} (m',\text{sb}',O')\)
- **assumes** less-sb: \(\forall i \in \bigcup (\text{fst }\text{ write-sops sb}). i < t\)
- **shows** \(\forall i \in \bigcup (\text{fst }\text{ write-sops sb'}). i < t\)

\(\langle\text{proof}\rangle\)

**lemma** issue-expr-load-tmps-range':
- \(\bigwedge t. \text{load-tmps} (\text{issue-expr t e}) = \{ i. t \leq i \land i < t + \text{used-tmps e}\} \)

\(\langle\text{proof}\rangle\)

**lemma** issue-expr-load-tmps-range:
- \(\bigwedge t. \forall i \in \text{load-tmps} (\text{issue-expr t e}). t \leq i \land i < t + (\text{used-tmps e})\)

\(\langle\text{proof}\rangle\)

**lemma** stmt-step-load-tmps-range':
- **assumes** step: \(\theta \vdash (s, t) \rightarrow_{s} ((s', t'),\text{is})\)
- **shows** \(\text{load-tmps} \text{ is} = \{ i. t \leq i \land i < t'\}\)

\(\langle\text{proof}\rangle\)

**lemma** stmt-step-load-tmps-range:
- **assumes** step: \(\theta \vdash (s, t) \rightarrow_{s} ((s', t'),\text{is})\)
- **shows** \(\forall i \in \text{load-tmps} \text{ is}. t \leq i \land i < t'\)

\(\langle\text{proof}\rangle\)

**lemma** distinct-load-tmps-issue-expr:
- \(\bigwedge t. \text{distinct-load-tmps} (\text{issue-expr t e})\)

\(\langle\text{proof}\rangle\)

**lemma** max-used-load-tmps:
- \(t + \text{used-tmps e} \notin \text{load-tmps} (\text{issue-expr t e})\)

\(\langle\text{proof}\rangle\)
lemma stmt-step-distinct-load-tmps:
  assumes step: \( \emptyset \vdash (s, t) \rightarrow_s (s', t'), is \)
  shows distinct-load-tmps is
  \langle proof \rangle

lemma store-sops-issue-expr [simp]: \( \forall t. \) store-sops (issue-expr t e) = {} 
  \langle proof \rangle

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 \( \forall (D, f) \in \) store-sops is. \( \forall i \in D. i < t' \)
  \langle proof \rangle

lemma sbh-step-distinct-load-tmps-prog-step:
  assumes step: \( \emptyset \vdash (s, t) \rightarrow_s (s', t'), is' \)
  assumes load-tmps-le: \( \forall i \in \) load-tmps is. \( i < t \)
  assumes read-tmps-le: \( \forall i \in \) 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) = {}
  \langle proof \rangle

lemma data-dependency-consistent-instrs-issue-expr:
  \( \forall t T. \) data-dependency-consistent-instrs T (issue-expr t e) 
  \langle proof \rangle

lemma dom-eval-expr:
  \( \forall t. \) \[ valid-sops-expr t e; x \in \) fst (eval-expr t e) \] \implies x \in \{i. i < t\} \cup load-tmps (issue-expr t e) 
  \langle proof \rangle

lemma Cond-not-s1: \( s_1 \neq \) Cond e s_1 s_2 
  \langle proof \rangle

lemma Cond-not-s2: \( s_2 \neq \) Cond e s_1 s_2 
  \langle proof \rangle

lemma Seq-not-s1: \( s_1 \neq \) Seq s_1 s_2 
  \langle proof \rangle

lemma Seq-not-s2: \( s_2 \neq \) Seq s_1 s_2 
  \langle proof \rangle

lemma prog-step-progress:
  assumes step: \( \emptyset \vdash (s, t) \rightarrow_s (s', t'), is \)
  shows \( (s', t') \neq (s, t) \lor \) is \( \neq \) [] 
  \langle proof \rangle
lemma stmt-step-data-dependency-consistent-instrs:
assumes step: θ ⊢ (s, t) → (s′, t′),is
assumes valid: valid-sops-stmt t s
shows data-dependency-consistent-instrs (\{ i. i < t \}) is
⟨proof⟩

lemma sbh-valid-data-dependency-prog-step:
assumes step: θ ⊢ (s, t) → (s′, t′),is
assumes store-sops-le: ∀ i ∈ Union (fst ' store-sops is). i < t
assumes write-sops-le: ∀ i ∈ Union (fst ' write-sops sb). i < t
assumes valid: valid-sops-stmt t s
shows data-dependency-consistent-instrs (\{ i. i < t \}) is′ ∧
load-tmps is′ ∩ Union (fst ' store-sops is) = \{ \} ∧
load-tmps is′ ∩ Union (fst ' write-sops sb) = \{ \}
⟨proof⟩

lemma sbh-load-tmps-fresh-prog-step:
assumes step: θ ⊢ (s, t) → (s′, t′),is
assumes tmps-le: ∀ i ∈ dom θ. i < t
shows load-tmps is′ ∩ dom θ = \{ \}
⟨proof⟩

lemma sbh-valid-sops-prog-step:
assumes step: θ ⊢ (s, t) → (s′, t′),is
assumes valid: valid-sops-stmt t s
shows ∀ sop ∈ store-sops is. valid-sop sop
⟨proof⟩

primrec prog-configs:: 'a memref list ⇒ 'a set
where
prog-configs [] = {}
prog-configs (x#xs) = (case x of
  Progsb pb p p′ is ⇒ \{ p, p′ \} ∪ prog-configs xs
| - ⇒ prog-configs xs)

lemma prog-configs-append: ∀ys. prog-configs (xs@ys) = prog-configs xs ∪ prog-configs ys
⟨proof⟩

lemma prog-configs-in1: Progsb p1 p2 is ∈ set xs ⇒ p1 ∈ prog-configs xs
⟨proof⟩

lemma prog-configs-in2: Progsb p1 p2 is ∈ set xs ⇒ p2 ∈ prog-configs xs
⟨proof⟩

lemma prog-configs-mono: ∀ys. set xs ⊆ set ys ⇒ prog-configs xs ⊆ prog-configs ys
⟨proof⟩
locale separated-tmps =
fixes ts
assumes valid-sops-stmt: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \] \[ \implies \text{valid-sops-stmt t s} \]
assumes valid-sops-stmt-sb: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O); (s', t') \in \text{prog-configs sb} \] \[ \implies \text{valid-sops-stmt t' s'} \]
assumes load-tmps-le: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \] \[ \implies \forall i \in \text{load-tmps is}. \ i < t \]
assumes read-tmps-le: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \] \[ \implies \forall i \in \text{read-tmps sb}. \ i < t \]
assumes store-sops-le: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \] \[ \implies \forall i \in \bigcup \text{fst ' store-sops is}. \ i < t \]
assumes write-sops-le: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \] \[ \implies \forall i \in \bigcup \text{fst ' write-sops sb}. \ i < t \]
assumes tmps-le: \[ i < \text{length ts}; \ \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \] \[ \implies \text{dom } \vartheta \cup \text{load-tmps is} = \{i. \ i < t\} \]

lemma (in separated-tmps)
\begin{enumerate}
\item ts-le':
\begin{enumerate}
\item assumes i-bound: \( i < \text{length ts} \)
\item assumes ts-i: \( \text{ts!i} = ((s, t), is, \vartheta, sb, D, O) \)
\item shows \( \forall i \in \text{dom } \vartheta. \ i < t \)
\end{enumerate}
\end{enumerate}
\langle proof \rangle

lemma (in separated-tmps) separated-tmps-nth-update:
\[ i < \text{length ts}; \ \text{valid-sops-stmt t s}; \ \forall (s', t') \in \text{prog-configs sb}. \ \text{valid-sops-stmt t' s'}; \] \[ \forall i \in \text{load-tmps is}. \ i < t; \forall i \in \text{read-tmps sb}. \ i < t; \] \[ \forall i \in \bigcup \text{fst ' store-sops is}. \ i < t; \forall i \in \bigcup \text{fst ' write-sops sb}. \ i < t; \] \[ \text{dom } \vartheta \cup \text{load-tmps is} = \{i. \ i < t\} \] \[ \implies \text{separated-tmps (ts[i:=((s, t), is, \vartheta, sb, D, O)])} \]
\langle proof \rangle

lemma hd-prog-app-in-first: \( \forall ys. \ \text{Prog}_{sb} p \ p' \is \in \text{set } xs \implies \text{hd-prog q (xs @ ys)} = \text{hd-prog q xs} \)
\langle proof \rangle

lemma hd-prog-app-in-eq: \( \forall ys. \ \text{Prog}_{sb} p \ p' \is \in \text{set } xs \implies \text{hd-prog q xs} = \text{hd-prog x xs} \)
\langle proof \rangle

lemma hd-prog-app-notin-first: \( \forall ys. \ \forall p \ p' \is. \ \text{Prog}_{sb} p \ p' \is \notin \text{set } xs \implies \text{hd-prog q (xs @ ys)} = \text{hd-prog q ys} \)
\langle proof \rangle

lemma union-eq-subsetD: \( A \cup B = C \implies A \cup B \subseteq C \land C \subseteq A \cup B \)
\langle proof \rangle

139
lemma prog-step-preserves-separated-tmps:
  assumes i-bound: \(i < \text{length } ts\)
  assumes ts-i: ts\(i = (p,\text{is},\theta,\text{sb},D,O)\)
  assumes prog-step: \(\theta \vdash p \rightarrow_s (p', \text{is}')\)
  assumes sep: separated-tmps ts
  shows separated-tmps
    \((ts \left[ i := (p', \text{is}@\text{is}', \theta, \text{sb}@[\text{Prog}_{sb} p p' \text{is}'], D, O) \right])\)
  \(\langle \text{proof} \rangle\)

lemma flush-step-sb-subset:
  assumes step: \((m, sb, O) \rightarrow_t (m', sb', O')\)
  shows set \(sb' \subseteq \text{set } sb\)
  \(\langle \text{proof} \rangle\)

lemma flush-step-preserves-separated-tmps:
  assumes i-bound: \(i < \text{length } ts\)
  assumes ts-i: ts\(i = (p,\text{is},\theta,\text{sb},D,O,\mathcal{R})\)
  assumes flush-step: \((m, sb, O, \mathcal{R}, \mathcal{S}) \rightarrow_t (m', sb', O', \mathcal{R}', \mathcal{S}')\)
  assumes sep: separated-tmps ts
  shows separated-tmps \((ts \left[ i := (p,\text{is},\theta,\text{sb},D,O,\mathcal{R}') \right])\)
  \(\langle \text{proof} \rangle\)

lemma sbh-step-preserves-store-sops-bound:
  assumes step: \((is, \theta, \text{sb}, m, D, O, \mathcal{R}, \mathcal{S}) \rightarrow_{sbh} (is', \theta', \text{sb}', m', D', O', \mathcal{R}', \mathcal{S}')\)
  assumes store-sops-le: \(\forall i \in \bigcup (\text{fst } \text{store-sops } \text{is}). i < t\)
  shows \(\forall i \in \bigcup (\text{fst } \text{store-sops } \text{is}'). i < t\)
  \(\langle \text{proof} \rangle\)

lemma sbh-step-preserves-write-sops-bound:
  assumes step: \((is, \theta, \text{sb}, m, D, O, \mathcal{R}, \mathcal{S}) \rightarrow_{sbh} (is', \theta', \text{sb}', m', D', O', \mathcal{R}', \mathcal{S}')\)
  assumes write-sops-le: \(\forall i \in \bigcup (\text{fst } \text{write-sops } \text{sb}). i < t\)
  shows \(\forall i \in \bigcup (\text{fst } \text{write-sops } \text{sb}'). i < t\)
  \(\langle \text{proof} \rangle\)

lemma sbh-step-prog-configs-eq:
  assumes step: \((is, \theta, \text{sb}, m, D, O, \mathcal{R}, \mathcal{S}) \rightarrow_{sbh} (is', \theta', \text{sb}', m', D', O', \mathcal{R}', \mathcal{S}')\)
  shows prog-configs \(sb' = \text{prog-configs } sb\)
  \(\langle \text{proof} \rangle\)

lemma sbh-step-preserves-tmps-bound:\:
  assumes step: \((is, \theta, \text{sb}, m, D, O, \mathcal{R}, \mathcal{S}) \rightarrow_{sbh} (is', \theta', \text{sb}', m', D', O', \mathcal{R}', \mathcal{S}')\)
  shows \(\text{dom } \theta \cup \text{load-tmps } \text{is} = \text{dom } \theta' \cup \text{load-tmps } \text{is}'\)
  \(\langle \text{proof} \rangle\)

lemma sbh-step-preserves-separated-tmps:
  assumes i-bound: \(i < \text{length } ts\)
  assumes ts-i: ts\(i = (p,\text{is},\theta,\text{sb},D,O,\mathcal{R})\)
  assumes memop-step: \((is, \theta, \text{sb}, m, D, O, \mathcal{R}, \mathcal{S}) \rightarrow_{sbh}\)
(is′, θ′, sb′, m′, D′, O′, R′, S′)
assumes instr: separated-tmps ts
shows separated-tmps (ts [i:= (p, is′, θ′, sb′, D′, O′, R′)])
(proof)

definition valid-pimp ts ≡ separated-tmps ts

lemma prog-step-preserves-valid:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb::stmt-config store-buffer, D, O, R)
assumes prog-step: θ ⊢ p →ₜ (p′, is′)
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:= (p′, is@is′, θ, sb@[Prog sb p p′ is′], D, O, R)])
(proof)

lemma flush-step-preserves-valid:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb::stmt-config store-buffer, D, O, R)
assumes flush-step: (m, sb, O, R, S) →ₜ (m′, sb′, O′, R′, S′)
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:= (p, is, θ, sb, D′, O′, R′)])
(proof)

lemma sbh-step-preserves-valid:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p, is, θ, sb::stmt-config store-buffer, D, O, R)
assumes memop-step: (is, θ, sb, m, D, O, R, S) → sbh
(is′, θ′, sb′, m′, D′, O′, R′, S′)
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:= (p, is′, θ′, sb′, D′, O′, R′)])
(proof)

lemma hd-prog-prog-configs: hd-prog p sb = p ∨ hd-prog p sb ∈ prog-configs sb
(proof)

interpretation PIMP: xvalid-program-progress stmt-step λ(s, t). valid-sops-stmt t s
valid-pimp
(proof)

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 λ p p′ is sb. () (proof)
interpretation virtual:
computation virtual-memop-step empty-storebuffer-step stmt-step λ p p′ is sb. () (proof)
interpretation store-buffer:
computation sb-memop-step store-buffer-step stmt-step λ p p′ is sb. sb (proof)
interpretation store-buffer-history:
computation sbh-memop-step flush-step stmt-step λp p’is sb. sb ⊙ [Prog sb p p’is]⟨proof⟩

abbreviation direct-pimp-step::
( stmt-config,unit,bool,owns,rels,shared) global-config ⇒
(- ⇒dp - [60,60] 100)

where
c ⇒dp d ≡ direct.concurrent-step c d

definition direct-pimp-steps::
( stmt-config,unit,bool,owns,rels,shared) global-config ⇒
(- ⇒dp ∗ - [60,60] 100)

where
direct-pimp-steps == direct-pimp-step ∗∗

Execution examples

lemma Assign-Const-ex:
([(Assign True (Tmp ({}),λθ. a)) (Const c) (λθ. A) (λθ. L) (λθ. R) (λθ. W))]
,(t),[],[],D,]{R],m,S) ⇒dp ∗
([(Skip,t),[],[],True,Θ ∪ A − R,Map.empty],m(a := c),S ⊕ W R ⊕ A L)
⟨proof⟩

lemma
([(Assign True (Tmp ({}),λθ. a)) (Binop (+) (Mem True x) (Mem True y)) (λθ. A) (λθ. L) (λθ. R) (λθ. W))]
,(t),[],[],D,]{R],m,S) ⇒dp ∗
([(Skip,t + 2),[],[],m x + t + 1 → m y),[],True,Θ ∪ A − R,Map.empty],m(a := m x + m y),S ⊕ W R ⊕ A L)
⟨proof⟩

lemma
assumes isTrue: isTrue c
shows
([(Cond (Const c) (Assign True (Tmp ({}),λθ. a)) (Const c) (λθ. A) (λθ. L) (λθ. R) (λθ. W))]
,(Skip,t),[],[],D,]{R],m,S) ⇒dp ∗
([(Skip,t),[],[],True,Θ ∪ A − R,Map.empty],m(a := c),S ⊕ W R ⊕ A L)
⟨proof⟩

end

References

4. Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, Types for Proofs and Programs, International Workshop, TYPES 2003,


