A Reduction Theorem for Store Buffers

Ernie Cohen[1], Norbert Schirmer[2,⋆]

[1] Microsoft Corp., Redmond, WA, USA
[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.

Table of Contents

1 Introduction

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

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

The simplest example illustrating such an inconsistency is the following program, consisting of two threads T0 and T1, where `x` and `y` are shared memory variables (initially 0) and `r0` and `r1` are registers:

```
T0          T1

x = 1;      y = 1;
r0 = y;     r1 = x;
```

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

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

In practice, the usual remedy for store buffering is adherence to a programming discipline that provides sequential consistency for a suitable class of architectures. In this report, we describe and prove the correctness of such a discipline suitable for the memory model provided by existing x86/x64 machines, where each write emerging from a store buffer hits a global cache visible to all processors. Because each processor sees the same global ordering of writes, this model is sometimes called *total store order* (TSO) [2][3]

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[4]). The main restriction on these accesses is that if a processor does a shared write and a

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

[4] This atomicity isn't guaranteed for certain memory types, or for operations that cross a cache line.

subsequent shared read (possibly from a different address), the processor must flush the store buffer somewhere in between. For example, in the example above, both `x` and `y` would be shared addresses, so each processor would have to flush its store buffer between its first and second operations.

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

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

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

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

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

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

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

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

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

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

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

2 Preliminaries

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

This article is written using Isabelle's document generation facilities, which guarantees a close correspondence between the presentation and the actual theory files. We distinguish formal entities typographically from other text. We use a sans serif font for types and constants (including functions and predicates), e.g., 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., $'a\ \mathsf{list}$ is the list type with type variable $'a$. In HOL all functions are total, e.g., $\mathsf{nat} \Rightarrow \mathsf{nat}$ is a total function on natural numbers. A function update is $f(y := v) = (\lambda x.\ \mathsf{if}\ x = y\ \mathsf{then}\ v\ \mathsf{else}\ f\ x)$. To formalize partial functions the type $'a\ \mathsf{option}$ is used. It is a data type with two constructors, one to inject values of the base type, e.g., $\lfloor x \rfloor$, and the additional element $\bot$. A base value can be projected with the function the, which is defined by the sole equation $\mathsf{the}\ \lfloor x \rfloor = x$. Since HOL is a total logic the term $\mathsf{the}\ \bot$ is still a well-defined yet un(der)specified value. Partial functions are usually represented by the type $'a \Rightarrow 'b\ \mathsf{option}$, abbreviated as $'a \rightharpoonup 'b$. They are commonly used as *maps*. We denote the domain of map m by $\mathsf{dom}\ m$. A map update is written as $m(a \mapsto v)$. We can restrict the domain of a map m to a set A by $m\restriction_A$.

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 $\mathsf{map}\ f\ xs$ 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 $\mathsf{dropWhile}\ P\ xs$ the prefix for which all elements satisfy predicate P are dropped from list xs.

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

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

3 Programming discipline

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

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

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

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

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

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

Table 1: Programming discipline.

(a) Access policy

	shared (read-write)	shared (read-only)	unshared
un-owned	vR, vW	vR, R	unreachable
owned	vR, vW, R	unreachable	vR, vW, R, W
owned by other	vR	unreachable	

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

(b) Flushing policy

	flush (before)
interlocked	as side effect
vR	if not clean
R, vW, W	never

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

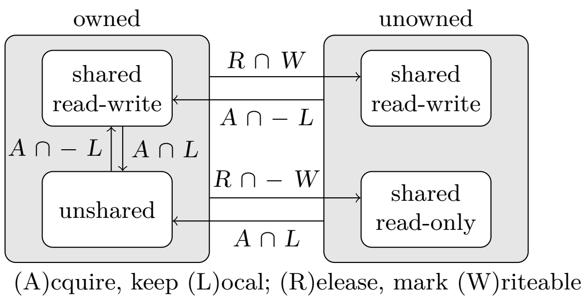

Fig. 1: Ownership transfer

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 j 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:

  $\mathsf{valid\text{-}sop}\ sop \equiv \forall\, D\ f\ j.\ sop = (D,\, f) \wedge D \subseteq \mathsf{dom}\ j \longrightarrow f\ j = f\ (j{\restriction}_D)$

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 $(p,\ is,\ j,\ sb)$ consisting of the program state p, a memory instruction list is, the map of temporaries j 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 $(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts',\ m')$ defined in Figure 2.

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ j,\ sb) \qquad j \vdash p \rightarrow_{\mathsf{p}} (p',\ is')}{(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts[i := (p',\ is\ @\ is',\ j,\ sb)],\ m)}$$

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ j,\ sb) \qquad (is,\ j,\ sb,\ m) \overset{\mathsf{sb}}{\rightarrow}_{\mathsf{m}} (is',\ j',\ sb',\ m')}{(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts[i := (p,\ is',\ j',\ sb')],\ m')}$$

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ j,\ sb) \qquad (m,\ sb) \rightarrow_{\mathsf{sb}} (m',\ sb')}{(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts[i := (p,\ is,\ j,\ sb')],\ m')}$$

Fig. 2: Global transitions of store buffer machine

A transition selects a thread $ts_{[i]} = (p,\ is,\ j,\ sb)$ 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 $j \vdash p \rightarrow_{\mathsf{p}} (p',\ is')$ takes the temporaries j 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 $(is,\ j,\ sb,\ m) \overset{\mathsf{sb}}{\rightarrow}_{\mathsf{m}} (is',\ j',\ sb',\ m')$ of a machine with store buffer may only fill its store buffer with new writes.

In a store buffer step $(m,\ sb) \rightarrow_{\mathsf{sb}} (m',\ sb')$ 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 $\text{WRITE}_{\mathsf{sb}}$ *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.

$$\frac{}{(m,\ \textsc{Write}_{\mathsf{sb}}\ \mathit{volatile}\ a\ \mathit{sop}\ v\ A\ L\ R\ W\ \#\ \mathit{sb}) \rightarrow_{\mathsf{sb}} (m(a := v),\ \mathit{sb})}$$

Fig. 3: Store buffer transition

$$\frac{v = (\mathtt{case}\ \mathsf{buffered\text{-}val}\ \mathit{sb}\ a\ \mathtt{of}\ \bot \Rightarrow m\ a \mid \lfloor v' \rfloor \Rightarrow v')}{(\textsc{Read}\ \mathit{volatile}\ a\ t\ \#\ \mathit{is},\ j,\ \mathit{sb},\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (\mathit{is},\ j(t \mapsto v),\ \mathit{sb},\ m)}$$

$$\frac{\mathit{sb}' = \mathit{sb}\ @\ [\textsc{Write}_{\mathsf{sb}}\ \mathit{volatile}\ a\ (D,\ f)\ (f\ j)\ A\ L\ R\ W]}{(\textsc{Write}\ \mathit{volatile}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ \mathit{sb},\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (\mathit{is},\ j,\ \mathit{sb}',\ m)}$$

$$\frac{\neg\ \mathit{cond}\ (j(t \mapsto m\ a)) \qquad j' = j(t \mapsto m\ a)}{(\mathrm{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ [],\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (\mathit{is},\ j',\ [],\ m)}$$

$$\frac{\mathit{cond}\ (j(t \mapsto m\ a)) \qquad j' = j(t \mapsto \mathit{ret}\ (m\ a)\ (f\ (j(t \mapsto m\ a)))) \qquad m' = m(a := f\ (j(t \mapsto m\ a)))}{(\mathrm{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ [],\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (\mathit{is},\ j',\ [],\ m')}$$

$$\frac{}{(\textsc{Fence}\ \#\ \mathit{is},\ j,\ [],\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (\mathit{is},\ j,\ [],\ m)}$$

$$\frac{}{(\textsc{Ghost}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ \mathit{sb},\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (\mathit{is},\ j,\ \mathit{sb},\ m)}$$

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 $(p,\ \mathit{is},\ j,\ \mathcal{D},\ \mathcal{O})$, with a dirty flag $\mathcal{D}$ indicating whether there may be an outstanding volatile write in the store buffer and the set of owned addresses $\mathcal{O}$. The dirty flag $\mathcal{D}$ 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 $(\mathit{ts},\ m,\ \mathcal{S})$ maintains a Boolean map of shared addresses $\mathcal{S}$ (indicating write permission). Addresses in the domain of mapping $\mathcal{S}$ are considered shared and the set of read-only addresses is obtained from $\mathcal{S}$ by: read-only $\mathcal{S} \equiv \{a.\ \mathcal{S}\ a = \lfloor \mathsf{False} \rfloor\}$

According to the rules in Fig 5 a global transition of the virtual machine $(\mathit{ts},\ m,\ \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow} (\mathit{ts}',\ m',\ \mathcal{S}')$ 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 $\mathcal{O}s,i\vdash (\mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd$ in Figure 7, where $\mathcal{O}s$ is the list of ownership sets obtained from the thread list ts and i is the index of the current

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ j,\ \mathcal{D},\ \mathcal{O}) \qquad j \vdash p \rightarrow_{\mathsf{p}} (p',\ is')}{(ts,\ m,\ \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow} (ts[i := (p',\ is\ @\ is',\ j,\ \mathcal{D},\ \mathcal{O})],\ m,\ \mathcal{S})}$$

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ j,\ \mathcal{D},\ \mathcal{O}) \qquad (is,\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S}) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is',\ j',\ m',\ \mathcal{D}',\ \mathcal{O}',\ \mathcal{S}')}{(ts,\ m,\ \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow} (ts[i := (p,\ is',\ j',\ \mathcal{D}',\ \mathcal{O}')],\ m',\ \mathcal{S}')}$$

Fig. 5: Global transitions of virtual machine

$$\frac{}{(\textsc{Read}\ volatile\ a\ t\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j(t \mapsto m\ a),\ x,\ m,\ ghst)}$$

$$\frac{}{(\textsc{Write}\ \mathsf{False}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ x,\ m(a := f\ j),\ ghst)}$$

$$\frac{ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad ghst' = (\mathsf{True},\ \mathcal{O} \cup A - R,\ \mathcal{S} \oplus_W R \ominus_A L)}{(\textsc{Write}\ \mathsf{True}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ x,\ m(a := f\ j),\ ghst')}$$

$$\frac{\neg\ cond\ (j(t \mapsto m\ a)) \qquad ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad ghst' = (\mathsf{False},\ \mathcal{O},\ \mathcal{S})}{(\mathrm{RMW}\ a\ t\ (D,\ f)\ cond\ ret\ A\ L\ R\ W\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j(t \mapsto m\ a),\ x,\ m,\ ghst')}$$

$$\frac{\begin{array}{c} cond\ (j(t \mapsto m\ a)) \qquad j' = j(t \mapsto ret\ (m\ a)\ (f\ (j(t \mapsto m\ a)))) \\ m' = m(a := f\ (j(t \mapsto m\ a))) \qquad ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad ghst' = (\mathsf{False},\ \mathcal{O} \cup A - R,\ \mathcal{S} \oplus_W R \ominus_A L) \end{array}}{(\mathrm{RMW}\ a\ t\ (D,\ f)\ cond\ ret\ A\ L\ R\ W\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j',\ x,\ m',\ ghst')}$$

$$\frac{ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad ghst' = (\mathsf{False},\ \mathcal{O},\ \mathcal{S})}{(\textsc{Fence}\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ x,\ m,\ ghst')}$$

$$\frac{ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad ghst' = (\mathcal{D},\ \mathcal{O} \cup A - R,\ \mathcal{S} \oplus_W R \ominus_A L)}{(\textsc{Ghost}\ A\ L\ R\ W\ \#\ is,\ j,\ x,\ m,\ ghst) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ x,\ m,\ ghst')}$$

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

$$\frac{a \in \mathcal{O} \vee a \in \textsf{read-only}\ \mathcal{S} \vee \textit{volatile} \wedge a \in \textsf{dom}\ \mathcal{S} \qquad \textit{volatile} \longrightarrow \neg\, \mathcal{D}}{\mathcal{O}s,i\vdash (\textsc{Read}\ \textit{volatile}\ a\ t \ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{a \in \mathcal{O} \qquad a \notin \textsf{dom}\ \mathcal{S}}{\mathcal{O}s,i\vdash (\textsc{Write}\ \textsf{False}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}\forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \qquad a \notin \textsf{read-only}\ \mathcal{S}\\ \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset \qquad A \subseteq \mathcal{O} \cup \textsf{dom}\ \mathcal{S} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset\end{array}}{\mathcal{O}s,i\vdash (\textsc{Write}\ \textsf{True}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\neg\ \textit{cond}\ (j(t \mapsto m\ a)) \qquad a \in \textsf{dom}\ \mathcal{S} \cup \mathcal{O}}{\mathcal{O}s,i\vdash (\textrm{RMW}\ a\ t\ (D,\ f)\ \textit{cond}\ \textit{ret}\ A\ L\ R\ W\ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}\textit{cond}\ (j(t \mapsto m\ a)) \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \qquad a \notin \textsf{read-only}\ \mathcal{S}\\ \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset \qquad A \subseteq \mathcal{O} \cup \textsf{dom}\ \mathcal{S} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset\end{array}}{\mathcal{O}s,i\vdash (\textrm{RMW}\ a\ t\ (D,\ f)\ \textit{cond}\ \textit{ret}\ A\ L\ R\ W\ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{}{\mathcal{O}s,i\vdash (\textsc{Fence}\ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{A \subseteq \textsf{dom}\ \mathcal{S} \cup \mathcal{O} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset}{\mathcal{O}s,i\vdash (\textsc{Ghost}\ A\ L\ R\ W\ \#\ \textit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

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: $\mathcal{O} \cup A - R$. 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: $\mathcal{S} \oplus_W R \ominus_A L$. The auxiliary ternary operators to augment and subtract addresses from the sharing map are defined as follows:

$\mathcal{S} \oplus_W R \equiv \lambda a.\ \textsf{if}\ a \in R\ \textsf{then}\ \lfloor a \in W \rfloor\ \textsf{else}\ \mathcal{S}\ a$

$\mathcal{S} \ominus_A L \equiv$
$\lambda a.\ \textsf{if}\ a \in L\ \textsf{then}\ \bot\ \textsf{else}\ \textsf{case}\ \mathcal{S}\ a\ \textsf{of}\ \bot \Rightarrow \bot \mid \lfloor \textit{writeable} \rfloor \Rightarrow \lfloor a \in A \vee \textit{writeable} \rfloor$

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. As the read-modify-write instruction is an interlocked operation which flushes the store buffer as a side effect the dirty flag $\mathcal{D}$ 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 $(ts_{\mathsf{sb}}, m) \overset{\mathsf{sb}}{\Rightarrow}^* (ts_{\mathsf{sb}}', m')$ can be simulated by a computation of the virtual machine $(ts, m, \mathcal{S})$ $\overset{\mathsf{v}}{\Rightarrow}^* (ts', m', \mathcal{S}')$. The main theorem only considers computations that start in an initial configuration where all store buffers are empty and end in a configuration where all store buffers are empty again. A configuration of the store buffer machine is obtained from a virtual configuration by removing all ghost components and assuming empty store buffers. This coupling relation between the thread configurations is written as $ts_{\mathsf{sb}} \sim ts$. Moreover, the precondition $\mathsf{initial_v}\ ts\ \mathcal{S}$ 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 $\mathcal{S}$) is unowned and every unowned address is shared. Finally with $\mathsf{safe\text{-}reach}\ (ts, m, S)$ 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).

$$\begin{array}{l}(ts_{\mathsf{sb}}, m) \overset{\mathsf{sb}}{\Rightarrow}^* (ts_{\mathsf{sb}}', m') \wedge \mathsf{empty\text{-}store\text{-}buffers}\ ts_{\mathsf{sb}}' \wedge ts_{\mathsf{sb}} \sim ts\ \wedge \mathsf{initial_v}\ ts\ \mathcal{S} \wedge \\ \quad \mathsf{safe\text{-}reach}\ (ts, m, \mathcal{S}) \longrightarrow \\ \quad\ \exists\, ts'\ \mathcal{S}'.\ (ts, m, \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow}^* (ts', m', \mathcal{S}') \wedge ts_{\mathsf{sb}}' \sim ts'\end{array}$$

This theorem captures our intiution 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 confiugrations 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 intructions 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 virutal 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 virutal machine below, which we try to motivate now. Esentially 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 (potenitally 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 $(is, j, m, \mathcal{D}, \mathcal{O}, \mathcal{R}, \mathcal{S}) \xrightarrow{\mathsf{v_d}}_{\mathsf{m}} (is', j', m', \mathcal{D}', \mathcal{O}', \mathcal{R}', \mathcal{S}')$ are defined Figure 8. The additional ghost component $\mathcal{R}$ is a mapping from addresses to a Boolean flag. If an address is in the domain of $\mathcal{R}$ 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 $\mathcal{R}$ 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:

$\mathsf{aug}\ (\mathsf{dom}\ \mathcal{S})\ R\ \mathcal{R} =$

$$\overline{(\textsc{Read}\ \mathit{volatile}\ a\ t\ \#\ \mathit{is},\ j,\ m,\ \mathit{ghst})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j(t \mapsto m\ a),\ m,\ \mathit{ghst})}$$

$$\overline{(\textsc{Write}\ \mathsf{False}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathit{ghst})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j,\ m(a := f\ j),\ \mathit{ghst})}$$

$$\frac{\mathit{ghst} = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad \mathit{ghst}' = (\mathsf{True},\ \mathcal{O} \cup A - R,\ \lambda x.\ \bot,\ \mathcal{S} \oplus_W R \ominus_A L)}{(\textsc{Write}\ \mathsf{True}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathit{ghst})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j,\ m(a := f\ j),\ \mathit{ghst}')}$$

$$\frac{\neg\ \mathit{cond}\ (j(t \mapsto m\ a)) \qquad \mathit{ghst} = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad \mathit{ghst}' = (\mathsf{False},\ \mathcal{O},\ \lambda x.\ \bot,\ \mathcal{S})}{(\mathrm{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathit{ghst})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j(t \mapsto m\ a),\ m,\ \mathit{ghst}')}$$

$$\frac{\begin{array}{c}\mathit{cond}\ (j(t \mapsto m\ a)) \qquad j' = j(t \mapsto \mathit{ret}\ (m\ a)\ (f\ (j(t \mapsto m\ a)))) \qquad m' = m(a := f\ (j(t \mapsto m\ a))) \\ \mathit{ghst} = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad \mathit{ghst}' = (\mathsf{False},\ \mathcal{O} \cup A - R,\ \lambda x.\ \bot,\ \mathcal{S} \oplus_W R \ominus_A L)\end{array}}{(\mathrm{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathit{ghst})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j',\ m',\ \mathit{ghst}')}$$

$$\overline{(\textsc{Fence}\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j,\ m,\ \mathsf{False},\ \mathcal{O},\ \lambda x.\ \bot,\ \mathcal{S})}$$

$$\frac{\mathit{ghst} = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad \mathit{ghst}' = (\mathcal{D},\ \mathcal{O} \cup A - R,\ \mathsf{aug}\ (\mathsf{dom}\ \mathcal{S})\ R\ \mathcal{R},\ \mathcal{S} \oplus_W R \ominus_A L)}{(\textsc{Ghost}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathit{ghst})\ \xrightarrow{\mathsf{vd}}_{\mathsf{m}}\ (\mathit{is},\ j,\ m,\ \mathit{ghst}')}$$

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

$(\lambda a.\ \mathsf{if}\ a \in R\ \mathsf{then\ case}\ \mathcal{R}\ a\ \mathsf{of}\ \bot \Rightarrow \lfloor a \in \mathsf{dom}\ \mathcal{S} \rfloor \mid \lfloor s \rfloor \Rightarrow \lfloor s \wedge a \in \mathsf{dom}\ \mathcal{S} \rfloor$
$\quad \mathsf{else}\ \mathcal{R}\ a)$

If an address is freshly released ($a \in R$ and $\mathcal{R}\ a = \bot$) the flag is set according to $\mathsf{dom}\ \mathcal{S}$. Otherwise the flag becomes $\lfloor \mathsf{False} \rfloor$ in case the released address is currently unshared. Note that with this definition $\mathcal{R}\ a = \lfloor \mathsf{False} \rfloor$ stays stable upon every new release and we do not loose information about a release of an unshared address.

The global transition $(ts,\ m,\ s) \overset{\mathsf{vd}}{\Rightarrow} (ts',\ m',\ s')$ are analogous to the rules in Figure 5 replacing the memory transtions with the refined version for delayed releases.

The safety judgment for delayed releases $\mathcal{O}s,\mathcal{R}s,i \vdash (\mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd$ is defined in Figure 9. Note the additional component $\mathcal{R}s$ 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 ($a \notin \mathsf{dom}\ \mathcal{R}s_{[j]}$); reading from an address is only allowed if it is not released by another thread while it was unshared ($\mathcal{R}s_{[j]}\ a \neq \lfloor \mathsf{False} \rfloor$).

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

- $\textsc{Read}_{\mathsf{sb}}\ \mathit{volatile}\ a\ t\ v$, recording a corresponding read from address a which loaded the value v to temporary t, or a
- $\textsc{Write}_{\mathsf{sb}}\ \mathit{volatile}\ a\ \mathit{sop}\ v$ for an outstanding write, where operation sop evaluated to value v, or of the form
- $\textsc{Prog}_{\mathsf{sb}}\ p\ p'\ \mathit{is}'$, recording a program transition from p to p' which issued instructions is', or of the form
- $\textsc{Ghost}_{\mathsf{sb}}\ A\ L\ R\ W$, recording a corresponding ghost operation.

As defined in Figure 10 a write updates the memory when it exits the store buffer, all other store buffer entries may only have an effect on the ghost state. The effect on the ownership

$$\frac{\begin{array}{c}a \in \mathcal{O} \vee a \in \textsf{read-only}\ \mathcal{S} \vee \mathit{volatile} \wedge a \in \textsf{dom}\ \mathcal{S} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow \mathcal{R}s_{[j]}\ a \neq \lfloor\textsf{False}\rfloor \\ \neg\ \mathit{volatile} \longrightarrow (\forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \textsf{dom}\ \mathcal{R}s_{[j]}) \qquad \mathit{volatile} \longrightarrow \neg\ \mathcal{D}\end{array}}{\mathcal{O}s,\mathcal{R}s,i \vdash (\textsc{Read}\ \mathit{volatile}\ a\ t\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{a \in \mathcal{O} \qquad a \notin \textsf{dom}\ \mathcal{S} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \textsf{dom}\ \mathcal{R}s_{[j]}}{\mathcal{O}s,\mathcal{R}s,i \vdash (\textsc{Write}\ \textsf{False}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}\forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \cup \textsf{dom}\ \mathcal{R}s_{[j]} \\ a \notin \textsf{read-only}\ \mathcal{S} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap (\mathcal{O}s_{[j]} \cup \textsf{dom}\ \mathcal{R}s_{[j]}) = \emptyset \\ A \subseteq \textsf{dom}\ \mathcal{S} \cup \mathcal{O} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset\end{array}}{\mathcal{O}s,\mathcal{R}s,i \vdash (\textsc{Write}\ \textsf{True}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\neg\ \mathit{cond}\ (j(t \mapsto m\ a)) \qquad a \in \textsf{dom}\ \mathcal{S} \cup \mathcal{O} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow \mathcal{R}s_{[j]}\ a \neq \lfloor\textsf{False}\rfloor}{\mathcal{O}s,\mathcal{R}s,i \vdash (\text{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}\mathit{cond}\ (j(t \mapsto m\ a)) \qquad a \in \textsf{dom}\ \mathcal{S} \cup \mathcal{O} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \cup \textsf{dom}\ \mathcal{R}s_{[j]} \\ a \notin \textsf{read-only}\ \mathcal{S} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap (\mathcal{O}s_{[j]} \cup \textsf{dom}\ \mathcal{R}s_{[j]}) = \emptyset \\ A \subseteq \textsf{dom}\ \mathcal{S} \cup \mathcal{O} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset\end{array}}{\mathcal{O}s,\mathcal{R}s,i \vdash (\text{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{}{\mathcal{O}s,\mathcal{R}s,i \vdash (\textsc{Fence}\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}A \subseteq \textsf{dom}\ \mathcal{S} \cup \mathcal{O} \\ L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap (\mathcal{O}s_{[j]} \cup \textsf{dom}\ \mathcal{R}s_{[j]}) = \emptyset\end{array}}{\begin{array}{c}\mathcal{O}s,\mathcal{R}s,i \vdash (\textsc{Ghost}\ A\ L\ R\ W\ \#\ \mathit{is},\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd \\ \mathcal{O}s,\mathcal{R}s,i \vdash ([],\ j,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd\end{array}}$$

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

$$\frac{}{(m,\ \textsc{Write}_{\textsf{sb}}\ \textsf{False}\ a\ \mathit{sop}\ v\ A\ L\ R\ W\ \#\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \rightarrow_{\textsf{sbh}} (m(a := v),\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S})}$$

$$\frac{\mathcal{O}' = \mathcal{O} \cup A - R \qquad \mathcal{S}' = \mathcal{S} \oplus_{W} R \ominus_{A} L}{(m,\ \textsc{Write}_{\textsf{sb}}\ \textsf{True}\ a\ \mathit{sop}\ v\ A\ L\ R\ W\ \#\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \rightarrow_{\textsf{sbh}} (m(a := v),\ \mathit{sb},\ \mathcal{O}',\ \lambda x.\ \bot,\ \mathcal{S}')}$$

$$\frac{}{(m,\ \textsc{Read}_{\textsf{sb}}\ \mathit{volatile}\ a\ t\ v\ \#\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \rightarrow_{\textsf{sbh}} (m,\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S})}$$

$$\frac{}{(m,\ \textsc{Prog}_{\textsf{sb}}\ p\ p'\ \mathit{is}\ \#\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \rightarrow_{\textsf{sbh}} (m,\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S})}$$

$$\frac{\mathcal{O}' = \mathcal{O} \cup A - R \qquad \mathcal{R}' = \textsf{aug}\ (\textsf{dom}\ \mathcal{S})\ R\ \mathcal{R} \qquad \mathcal{S}' = \mathcal{S} \oplus_{W} R \ominus_{A} L}{(m,\ \textsc{Ghost}_{\textsf{sb}}\ A\ L\ R\ W\ \#\ \mathit{sb},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \rightarrow_{\textsf{sbh}} (m,\ \mathit{sb},\ \mathcal{O}',\ \mathcal{R}',\ \mathcal{S}')}$$

Fig. 10: Store buffer transitions with history

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

$$\frac{v = (\mathsf{case\ buffered\text{-}val}\ sb\ a\ \mathsf{of}\ \bot \Rightarrow m\ a \mid \lfloor v' \rfloor \Rightarrow v') \qquad sb' = sb\ @\ [\textsc{Read}_{\mathsf{sb}}\ volatile\ a\ t\ v]}{(\textsc{Read}\ volatile\ a\ t\ \#\ is,\ j,\ sb,\ m,\ ghst) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j(t \mapsto v),\ sb',\ m,\ ghst)}$$

$$\frac{sb' = sb\ @\ [\textsc{Write}_{\mathsf{sb}}\ \mathsf{False}\ a\ (D,\ f)\ (f\ j)\ A\ L\ R\ W]}{(\textsc{Write}\ \mathsf{False}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ is,\ j,\ sb,\ m,\ ghst) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ sb',\ m,\ ghst)}$$

$$\frac{\begin{array}{c} sb' = sb\ @\ [\textsc{Write}_{\mathsf{sb}}\ \mathsf{True}\ a\ (D,\ f)\ (f\ j)\ A\ L\ R\ W] \\ ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad ghst' = (\mathsf{True},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \end{array}}{(\textsc{Write}\ \mathsf{True}\ a\ (D,\ f)\ A\ L\ R\ W\ \#\ is,\ j,\ sb,\ m,\ ghst) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ sb',\ m,\ ghst')}$$

$$\frac{\neg\ cond\ (j(t \mapsto m\ a)) \qquad ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad ghst' = (\mathsf{False},\ \mathcal{O},\ \lambda x.\ \bot,\ \mathcal{S})}{(\textrm{RMW}\ a\ t\ (D,\ f)\ cond\ ret\ A\ L\ R\ W\ \#\ is,\ j,\ [],\ m,\ ghst) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j(t \mapsto m\ a),\ [],\ m,\ ghst')}$$

$$\frac{\begin{array}{c} cond\ (j(t \mapsto m\ a)) \qquad j' = j(t \mapsto ret\ (m\ a)\ (f\ (j(t \mapsto m\ a)))) \qquad m' = m(a := f\ (j(t \mapsto m\ a))) \\ ghst = (\mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \qquad ghst' = (\mathsf{False},\ \mathcal{O} \cup A - R,\ \lambda x.\ \bot,\ \mathcal{S} \oplus_{\mathsf{W}} R \ominus_{\mathsf{A}} L) \end{array}}{(\textrm{RMW}\ a\ t\ (D,\ f)\ cond\ ret\ A\ L\ R\ W\ \#\ is,\ j,\ [],\ m,\ ghst) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j',\ [],\ m',\ ghst')}$$

$$\frac{}{(\textsc{Fence}\ \#\ is,\ j,\ [],\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{R},\ \mathcal{S}) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ [],\ m,\ \mathsf{False},\ \mathcal{O},\ \lambda x.\ \bot,\ \mathcal{S})}$$

$$\frac{}{(\textsc{Ghost}\ A\ L\ R\ W\ \#\ is,\ j,\ sb,\ m,\ G) \overset{\mathsf{sbh}}{\rightarrow}_{\mathsf{m}} (is,\ j,\ sb\ @\ [\textsc{Ghost}_{\mathsf{sb}}\ A\ L\ R\ W],\ m,\ G)}$$

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

are analogous to the rules in Figure 2 replacing the memory transtions and store buffer transtiontions 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 histroy and the virtual machine with delayed releases. Remember the basic idea of the coupling relation: the state of the virtual machine is obtained from the state of the store buffer machine, by executing each store buffer until we reach the first volatile write. The remaining store buffer entries are suspended as instructions. The instructions now also include the history entries for reads, program steps and ghost operations. The suspended reads are not yet visible in the temporaries of the virtual machine. Similar the ownership effects (and program steps) of the suspended operations are not yet visible in the virtual machine. The coupling relation between the store buffer machine and the virtual machine is illustrated in Figure 12. The threads issue instructions to the store buffers from the right and the instructions emerge from the store buffers to main memory from the left. The dotted line illustrates the state of the virtual machines memory. It is obtained from the memory of the store buffer machine by executing the purely non-volatile prefixes of the store buffers. The remaining entries of the store buffer are still (suspended) instructions in the virtual machine.

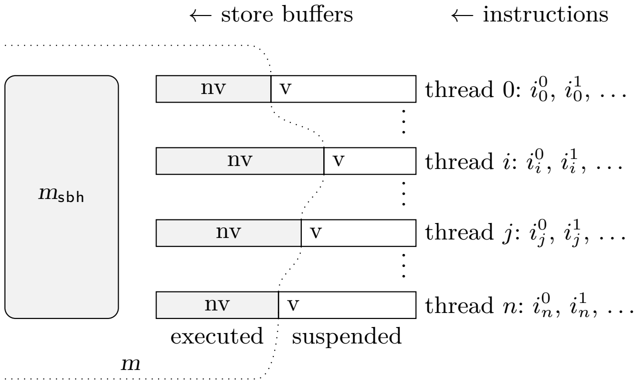

Fig. 12: Illustration of coupling relation

Consider the following configuration of a thread $ts_{\mathsf{sbh}[j]}$ 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.

$$ts_{\mathsf{sbh}[j]} = (p, [i_1, \ldots, i_{\mathsf{n}}], j, [s_1, \ldots, s_{\mathsf{v}}, s_{\mathsf{v}+1}, \ldots, s_{\mathsf{m}}], \mathcal{D}, \mathcal{O}, \mathcal{R})$$

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 $\textsc{Read}_{\mathsf{sb}}$ / $\textsc{Write}_{\mathsf{sb}}$ / $\textsc{Ghost}_{\mathsf{sb}}$ is converted to a $\textsc{Read}$ / $\textsc{Write}$ / $\textsc{Ghost}$ 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 j'. Similar, the suspended updates to the ownership sets and dirty flag are not yet recorded in $\mathcal{O}'$, $\mathcal{R}'$ and $\mathcal{D}'$.

$$ts_{[j]} = (p, [s_{\mathsf{v}}, s_{\mathsf{v}+1}, \ldots, s_{\mathsf{m}}, i_1, \ldots, i_{\mathsf{n}}], j', \mathcal{D}', \mathcal{O}', \mathcal{R}')$$

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:

$$ts_{\mathsf{sbh}}{}'_{[j]} = (p', [i_1, \ldots, i_{\mathsf{n}}] \mathbin{@} is', j, [s_1, \ldots, s_{\mathsf{v}}, \ldots, s_{\mathsf{m}}, \textsc{Prog}_{\mathsf{sb}}\ p\ p'\ is'], \mathcal{D}, \mathcal{O}, \mathcal{R})$$

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 $(ts_{\mathsf{sbh}}, m_{\mathsf{sbh}}, \mathcal{S}_{\mathsf{sbh}}) \sim (ts, m, \mathcal{S})$, 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

$$
\begin{array}{l}
\qquad\qquad m = \textsf{exec-all-until-volatile-write}\ ts_{\textsf{sbh}}\ m_{\textsf{sbh}} \\
\qquad \mathcal{S} = \textsf{share-all-until-volatile-write}\ ts_{\textsf{sbh}}\ \mathcal{S}_{\textsf{sbh}} \qquad |ts_{\textsf{sbh}}| = |ts| \\
\forall\, i{<}|ts_{\textsf{sbh}}|. \\
\quad \texttt{let}\ (p_{\textsf{sbh}},\ is_{\textsf{sbh}},\ j_{\textsf{sbh}},\ sb,\ \mathcal{D}_{\textsf{sbh}},\ \mathcal{O}_{\textsf{sbh}},\ \mathcal{R}_{\textsf{sbh}}) = ts_{\textsf{sbh}[i]}; \\
\qquad execs = \textsf{takeWhile not-volatile-write}\ sb; \\
\qquad suspends = \textsf{dropWhile not-volatile-write}\ sb \\
\quad \texttt{in}\ \exists\, is\ \mathcal{D}.\ \textsf{instrs}\ suspends\ @\ is_{\textsf{sbh}} = is\ @\ \textsf{prog-instrs}\ suspends\ \wedge \\
\qquad\qquad \mathcal{D}_{\textsf{sbh}} = (\mathcal{D} \vee \textsf{refs volatile-Write}\ sb \neq \emptyset)\ \wedge \\
\qquad\qquad ts_{[i]} = \\
\qquad\qquad (\textsf{hd-prog}\ p_{\textsf{sbh}}\ suspends,\ is,\ j_{\textsf{sbh}}{\restriction}_{(-\ \textsf{read-tmps}\ suspends)},\ \mathcal{D}, \\
\qquad\qquad\ \textsf{acquire}\ execs\ \mathcal{O}_{\textsf{sbh}},\ \textsf{release}\ execs\ (\textsf{dom}\ \mathcal{S}_{\textsf{sbh}})\ \mathcal{R}_{\textsf{sbh}}) \\
\hline
\qquad\qquad (ts_{\textsf{sbh}},\ m_{\textsf{sbh}},\ \mathcal{S}_{\textsf{sbh}}) \sim (ts,\ m,\ \mathcal{S})
\end{array}
$$

Fig. 13: Coupling relation

machine. We collect the additional instructions which were issued by program instructions but still recorded in the remainder of the store buffer with function prog-instrs. These instructions have already made their way to the instructions of the store buffer machine but not yet on the virtual machine. This situation is formalized as $\textsf{instrs}\ suspends\ @\ is_{\textsf{sbh}} = is\ @\ \textsf{prog-instrs}\ suspends$, where 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 hd-prog. The temporaries of the virtual machine are obtained by removing the suspended reads from j. 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 exec-all-until-volatile-write executes them in order of appearance. Similarly the sharing map of the virtual machine is obtained by executing all store buffers until the first volatile write via the function share-all-until-volatile-write. For the local ownership set $\mathcal{O}_{\textsf{sbh}}$ the auxiliary function acquire calculates the outstanding effect of the already simulated parts of the store buffer. Analogously release calculates the effect for the released addresses $\mathcal{R}_{\textsf{sbh}}$.

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).

$$
\begin{array}{l}
(ts_{\textsf{sbh}},\ m_{\textsf{sbh}},\ \mathcal{S}_{\textsf{sbh}}) \overset{\textsf{sbh}}{\Rightarrow} (ts_{\textsf{sbh}}',\ m_{\textsf{sbh}}',\ \mathcal{S}_{\textsf{sbh}}') \wedge (ts_{\textsf{sbh}},\ m_{\textsf{sbh}},\ \mathcal{S}_{\textsf{sbh}}) \sim (ts,\ m,\ \mathcal{S}) \wedge \\
\ \ \textsf{safe-reach-delayed}\ (ts,\ m,\ \mathcal{S}) \wedge \textsf{invariant}\ ts_{\textsf{sbh}}\ \mathcal{S}_{\textsf{sbh}}\ m_{\textsf{sbh}} \longrightarrow \\
\ \ \ \textsf{invariant}\ ts_{\textsf{sbh}}'\ \mathcal{S}_{\textsf{sbh}}'\ m_{\textsf{sbh}}' \wedge \\
\ \ (\exists\, ts'\ \mathcal{S}'\ m'.\ (ts,\ m,\ \mathcal{S}) \overset{\textsf{vd}}{\Rightarrow}^{*} (ts',\ m',\ \mathcal{S}') \wedge (ts_{\textsf{sbh}}',\ m_{\textsf{sbh}}',\ \mathcal{S}_{\textsf{sbh}}') \sim (ts',\ m',\ \mathcal{S}'))
\end{array}
$$

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

$$\textsf{invariant}\ ts_{\textsf{sbh}}\ \mathcal{S}_{\textsf{sbh}}\ m_{\textsf{sbh}} \equiv \textsf{ownership-inv}\ \mathcal{S}_{\textsf{sbh}}\ ts_{\textsf{sbh}} \wedge \textsf{sharing-inv}\ \mathcal{S}_{\textsf{sbh}}\ ts_{\textsf{sbh}} \wedge$$

$\mathsf{temporaries\text{-}inv}\ ts_{\mathsf{sbh}} \wedge \mathsf{data\text{-}dependency\text{-}inv}\ ts_{\mathsf{sbh}} \wedge \mathsf{history\text{-}inv}\ ts_{\mathsf{sbh}}\ m_{\mathsf{sbh}} \wedge \mathsf{flush\text{-}inv}\ ts_{\mathsf{sbh}} \wedge$
$\mathit{valid}\ ts_{\mathsf{sbh}}$

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

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

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

Data dependency. Data dependency means that store operations may only depend on *previous* read operations. For every thread we have: (i) Every operation $(D,\ f)$ in a write instruction or a store buffer write is valid according to valid-sop $(D,\ f)$, i.e., function f only depends on domain D. (ii) For every suffix of the instructions of the form $\textsc{Write}$ *volatile* $a\ (D,\ f)\ A\ L\ R\ W\ \#\ \mathit{is}$ 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 $\textsc{Write}_{\mathsf{sb}}$ *volatile* $a\ (D,\ f)\ v\ A\ L\ R\ W$ the recorded value v coincides with $f\ j$, and domain D is subset of dom j 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 $\textsc{Prog}_{\mathsf{sb}}\ p_1\ p_2\ \mathit{is}'\ \#\ sb'$, 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 $j{\restriction}_{(-\ \mathsf{read\text{-}tmps}\ sb')} \vdash p_1 \rightarrow_{\mathsf{p}} (p_2,\ \mathit{is}')$ 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: $\exists\, \mathit{is}'.\ \mathsf{instrs}\ sb'\ @\ \mathit{is} = \mathit{is}'\ @\ \mathsf{prog\text{-}instrs}\ sb'$, i.e., the remaining instructions *is* correspond to a suffix of the recorded instructions prog-instrs sb'.

Flushes. If the dirty flag is unset there are no outstanding volatile writes in the store buffer.

Program step. The program-transitions are still a parameter of our model. In order to make the proof work, we have to assume some of the invariants also for the program steps. We allow the program-transitions to employ further invariants on the configurations, these are modeled by the parameter *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 *j* 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 $c_{\mathsf{sbh}} = (ts_{\mathsf{sbh}}, m_{\mathsf{sbh}}, \mathcal{S}_{\mathsf{sbh}})$ of the store buffer machine, where the next instruction to be executed is a read of thread i: $\textsc{Read}_{\mathsf{sb}}$ *volatile* a t. The configuration of the virtual machine is $cfg = (ts, m, \mathcal{S})$. We have to simulate this step on the virtual machine and can make use of the coupling relations $(ts_{\mathsf{sbh}}, m_{\mathsf{sbh}}, \mathcal{S}_{\mathsf{sbh}}) \sim (ts, m, \mathcal{S})$, the invariants invariant ts_{sbh} $\mathcal{S}_{\mathsf{sbh}}$ m_{sbh} and the safety of all reachable states of the virtual machine: safe-reach-delayed $(ts, m, \mathcal{S})$. 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 machien 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 $cfg'' = (ts'', m'', \mathcal{S}'')$, 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 $v = m_{\mathsf{sbh}}\ a$. Moreover, we make a case distinction wheter there is an outstanding volatile write in the store buffer of thread i or

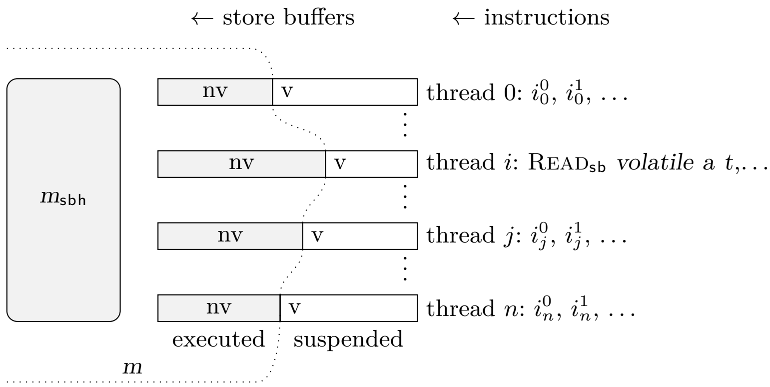

Fig. 14: Thread i poised to read

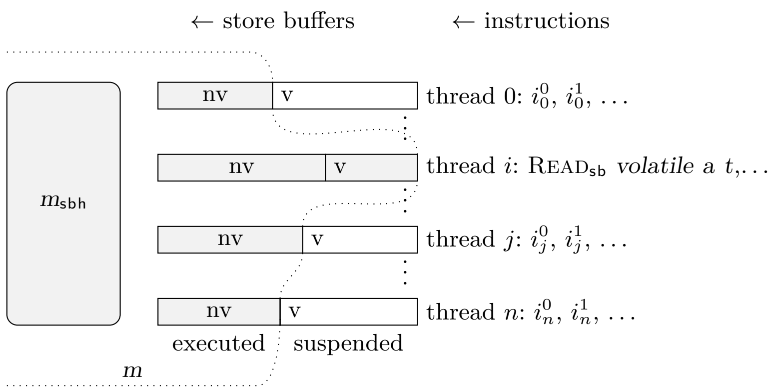

Fig. 15: Forwarded computation of virtual machine

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 $cfg'' = cfg$. We have to show that the virtual machine reads the same value from memory: $v = m\ a$. So what can go wrong? When can the the memory of the virtual machine hold a different value? The memory of the virtual machine is obtained from the memory of the store buffer machine by executing all store buffers until we hit the first volatile write. So if there is a discrepancy in the value this has to come from a non-volatile write in the executed parts of another thread, let us say thread j. This write is marked as x in Figure 16.

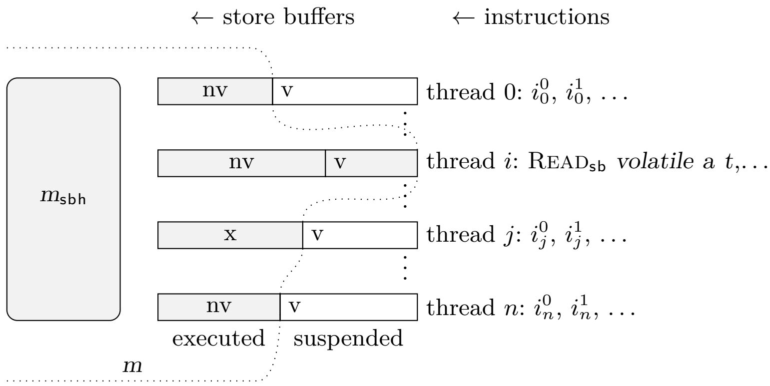

Fig. 16: Conflicting write in thread j (marked x)

We refer to x both for the write operation itself and to characterize the point in time in the computation of the virtual machine where the write was executed. At the point x the write was safe according to rules in Figure 9 for non-volatile writes. So it was owned by thread j and unshared. This knowledge about the safety of write x is preserved in the invariants, namely (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 cfg: address a either has to be owned by thread i, or has to be read-only or the read is volatile and a is shared. Additionally there are the constraints on the released addresses which we will exploit below. Let us address all cases step by step. First, we consider that address a is currently owned by thread i. As it was owned by thread j at time x there has to be an release of a in the executed prefix of the store buffer of thread j. This release is recorded in the release set, so we know $a \in \mathsf{dom}\ \mathcal{R}s_{[j]}$. This contradicts the safety of the read. Second, we consider that address a is currently read-only. At time x address a was owned by thread j, unshared and not read-only. Hence there was a release of address a in the executed prefix of the store buffer of j, where it made a transition unshared and owned to shared. With the monotonicity of the release sets this means $a \in \mathsf{dom}\ \mathcal{R}s_{[j]}$, even more precisely $\mathcal{R}s_{[j]}\ a = \lfloor \mathsf{False} \rfloor$. Hence there is no chance to get the read safe (neiter a volatile nor a non-volatile). Third, consider a volatile read and that address a is currently shared. This is ruled out by the same line of reasoning as in the previous case. So ultimately we have ruled out all races that could destroy the value at address a and have shown that we can simulate the step on the virtual machine. This completes the simulation of the case where there is no store buffer forwarding and no volatile write in the store buffer of thread i. The other cases are handled similar. The main arguments are obtained by arguing about safety of configuration cfg'' and exploiting the invariants to rule out conflicting operations in other store buffers. When there is a volatile write in he 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. Besided 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 cfg''. However, consider an ghost operation that acquires an address a. From safety of the configuration cfg'' we can only infer that there is no conflicting acquire in the non-volaitle prefixes of the other store buffers. In case an conflicting acquire is in the suspended part of a store buffer of thread j safety of configuration cfg'' is not enough. But as we have safety of all reachable states we can forward the computation of thread j until the conflicting acquire is about to be executed and construct an unsafe state which rules out the conflict.

Last we want to comment on the case where the store buffer takes a step. The major case destinction is wheter a volatile write leaves the store buffer or not. In the former case the virtual machine has to simulate a whole bunch of instructions at once to simulate the store buffer machine up to the next volatile write in the store buffer. In the latter case the virtual machine does no step at all, since the instruction leaving the store buffer is already simulated. In both cases one key argument is commutativity of non-volatile operations with respect to global effects on the memory or the sharing map. Consider a non-volatile store buffer step of thread i. In the configuration of the virtual machine before the store buffer step of thread i, the simulation relation applies the update to the memory and the sharing map of the store buffer machine, within the operations exec-all-until-volatile-write and 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 conludes the discussion on the proof of Theorem 2. □

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

1. simulate the computation of the store buffer machine $(ts_{\mathsf{sb}}, m) \overset{\mathsf{sb}}{\Rightarrow}^{*} (ts_{\mathsf{sb}}', m')$ by a computation of a store buffer machine with history $(ts_{\mathsf{sbh}}, m, \mathcal{S}) \overset{\mathsf{sbh}}{\Rightarrow}^{*} (ts_{\mathsf{sbh}}', m', \mathcal{S}')$,
2. simulate the computation of the store buffer machine with history by a computation of the virtual machine with delayed releases $(ts, m, \mathcal{S}) \overset{\mathsf{v_d}}{\Rightarrow}^{*} (ts', m', \mathcal{S}')$ 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 $(ts, m, \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow}^{*} (ts', m', \mathcal{S}')$[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 $(ts, m, \mathcal{S})$ provided by the preconditions of Theorem 1 to the required safe-reach-delayed $(ts, m, \mathcal{S})$. 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 $(ts, m, \mathcal{S})$ 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 simulataneously be unsafe with respect to free flowing releases). Now we aim to find a configuration c' that is also reachable from the initial configuration and is unsafe with respect to free flowing releases. Intuitively this is a configuration where Thread i is rewinded to the state just before the release of address a and Thread j is in the same state as in configuration c. Before the release of a the address has to be owned by Thread i, which is unsafe according to free flowing releases as well as delayed releases. So we can argue that either Thread j can reach the same state although Thread i is rewinded or we even hit an unsafe configuration before. What kind of steps can Thread i perform between between the free flowing release point (point of the ghost instruction) and the delayed release point (point of next volatile write, interlocked operation or fence at which the release map is emptied)? How can these actions affect Thread j? Note that the delayed release point is not yet reached as this would empty the release map (which we know not to be empty). Thus Thread i does only perform reads, ghost instructions, program steps or non-volatile writes. All of these instructions of Thread i either have no influence on the computation of Thread j at all (e.g. a read, program step, non-volatile write or irrelevant ghost operation) or may cause a safety violation already in a shorter computation (e.g. acquiring an address that another thread holds). This is fine for our inductive argument. So either we can replay every step of Thread j and reach the final configuration c' which is now also unsafe according to free flowing releases, or we hit a configuration c'' in a shorter computation which violates the rules of delayed as well as free flowing releases (using the induction hypothesis).

6 PIMP

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

An expression e is either (i) Const v, a constant value, (ii) Mem *volatile* a, a (volatile) memory lookup at address a, (iii) Tmp *sop*, reading from the temporaries with a operation *sop* which is an intermediate expression occurring in the transition rules for statements,

(iv) $\textsc{Unop}\ f\ e$, a unary operation where f is a unary function on values, and finally (v) $\textsc{Binop}\ f\ e_1\ e_2$, a binary operation where f is a binary function on values.

A statement s is either (i) $\textsc{Skip}$, the empty statement, (ii) $\textsc{Assign}\ \mathit{volatile}\ a\ e\ A\ L\ R\ W$, a (volatile) assignment of expression e to address expression a, (iii) CAS $a\ c_\mathsf{e}\ s_\mathsf{e}\ A\ L\ R\ W$, atomic compare and swap at address expression a with compare expression c_e and swap expression s_e, (iv) $\textsc{Seq}\ s_1\ s_2$, sequential composition, (v) $\textsc{Cond}\ e\ s_1\ s_2$, the if-then-else statement, (vi) $\textsc{While}\ e\ s$, the loop statement with condition e, (vii) S$\textsc{Ghost}$, and S$\textsc{Fence}$ 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 $\mathsf{used\text{-}tmps}\ e$ calculates

$$
\begin{array}{lcl}
\mathsf{issue\text{-}expr}\ t\ (\textsc{Const}\ v) & = & [] \\
\mathsf{issue\text{-}expr}\ t\ (\textsc{Mem}\ \mathit{volatile}\ a) & = & [\textsc{Read}\ \mathit{volatile}\ a\ t] \\
\mathsf{issue\text{-}expr}\ t\ (\textsc{Tmp}\ (D,\ f)) & = & [] \\
\mathsf{issue\text{-}expr}\ t\ (\textsc{Unop}\ f\ e) & = & \mathsf{issue\text{-}expr}\ t\ e \\
\mathsf{issue\text{-}expr}\ t\ (\textsc{Binop}\ f\ e_1\ e_2) & = & \mathsf{issue\text{-}expr}\ t\ e_1\ @\ \mathsf{issue\text{-}expr}\ (t + \mathsf{used\text{-}tmps}\ e_1)\ e_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{eval\text{-}expr}\ t\ (\textsc{Const}\ v) & = & (\emptyset,\ \lambda j.\ v) \\
\mathsf{eval\text{-}expr}\ t\ (\textsc{Mem}\ \mathit{volatile}\ a) & = & (\{t\},\ \lambda j.\ \mathsf{the}\ (j\ t)) \\
\mathsf{eval\text{-}expr}\ t\ (\textsc{Tmp}\ (D,\ f)) & = & (D,\ f) \\
\mathsf{eval\text{-}expr}\ t\ (\textsc{Unop}\ f\ e) & = & \mathtt{let}\ (D,\ f_\mathsf{e}) = \mathsf{eval\text{-}expr}\ t\ e\ \mathtt{in}\ (D,\ \lambda j.\ f\ (f_\mathsf{e}\ j)) \\
\mathsf{eval\text{-}expr}\ t\ (\textsc{Binop}\ f\ e_1\ e_2) & = & \mathtt{let}\ (D_1,\ f_1) = \mathsf{eval\text{-}expr}\ t\ e_1; \\
 & & \qquad (D_2,\ f_2) = \mathsf{eval\text{-}expr}\ (t + \mathsf{used\text{-}tmps}\ e_1)\ e_2 \\
 & & \mathtt{in}\ (D_1 \cup D_2,\ \lambda j.\ f\ (f_1\ j)\ (f_2\ j))
\end{array}
$$

Fig. 17: Expression evaluation

the number of temporaries that are necessary to evaluate expression e, where every $\textsc{Mem}$ expression accounts to one temporary. With $\mathsf{issue\text{-}expr}\ t\ e$ we obtain the instruction list for expression e starting at temporary t, whereas $\mathsf{eval\text{-}expr}\ t\ e$ 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_a if successful. In either case the temporary finally returns the old value read.

Sequential composition is straightforward. An if-then-else is computed by first issuing the memory instructions for evaluation of condition e and transforming the condition to an operation on temporaries. When the result is available the transition to the first or second statement is made, depending on the result of isTrue. Execution of the loop is defined

$$\frac{\forall\, \mathit{sop}.\ a \neq \textsc{Tmp}\ \mathit{sop} \qquad a' = \textsc{Tmp}\ (\mathsf{eval\text{-}expr}\ t\ a) \qquad t' = t + \mathsf{used\text{-}tmps}\ a \qquad \mathit{is} = \mathsf{issue\text{-}expr}\ t\ a}{j \vdash (\textsc{Assign}\ \mathit{volatile}\ a\ e\ A\ L\ R\ W,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Assign}\ \mathit{volatile}\ a'\ e\ A\ L\ R\ W,\ t'),\ \mathit{is})}$$

$$\frac{D \subseteq \mathsf{dom}\ j \qquad \mathit{is} = \mathsf{issue\text{-}expr}\ t\ e\ @\ [\textsc{Write}\ \mathit{volatile}\ (a\ j)\ (\mathsf{eval\text{-}expr}\ t\ e)\ (A\ j)\ (L\ j)\ (R\ j)\ (W\ j)]}{j \vdash (\textsc{Assign}\ \mathit{volatile}\ (\textsc{Tmp}\ (D,\ a))\ e\ A\ L\ R\ W,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Skip},\ t + \mathsf{used\text{-}tmps}\ e),\ \mathit{is})}$$

$$\frac{\forall\, \mathit{sop}.\ a \neq \textsc{Tmp}\ \mathit{sop} \qquad a' = \textsc{Tmp}\ (\mathsf{eval\text{-}expr}\ t\ a) \qquad t' = t + \mathsf{used\text{-}tmps}\ a \qquad \mathit{is} = \mathsf{issue\text{-}expr}\ t\ a}{j \vdash (\mathrm{CAS}\ a\ c_{\mathsf{e}}\ s_{\mathsf{e}}\ A\ L\ R\ W,\ t) \rightarrow_{\mathsf{p}} ((\mathrm{CAS}\ a'\ c_{\mathsf{e}}\ s_{\mathsf{e}}\ A\ L\ R\ W,\ t'),\ \mathit{is})}$$

$$\frac{\forall\, \mathit{sop}.\ c_{\mathsf{e}} \neq \textsc{Tmp}\ \mathit{sop} \qquad {c_{\mathsf{e}}}' = \textsc{Tmp}\ (\mathsf{eval\text{-}expr}\ t\ c_{\mathsf{e}}) \qquad t' = t + \mathsf{used\text{-}tmps}\ c_{\mathsf{e}} \qquad \mathit{is} = \mathsf{issue\text{-}expr}\ t\ c_{\mathsf{e}}}{j \vdash (\mathrm{CAS}\ (\textsc{Tmp}\ a)\ c_{\mathsf{e}}\ s_{\mathsf{e}}\ A\ L\ R\ W,\ t) \rightarrow_{\mathsf{p}} ((\mathrm{CAS}\ (\textsc{Tmp}\ a)\ {c_{\mathsf{e}}}'\ s_{\mathsf{e}}\ A\ L\ R\ W,\ t'),\ \mathit{is})}$$

$$\frac{\begin{array}{c} D_{\mathsf{a}} \subseteq \mathsf{dom}\ j \\ D_{\mathsf{c}} \subseteq \mathsf{dom}\ j \qquad \mathsf{eval\text{-}expr}\ t\ s_{\mathsf{e}} = (D,\ f) \qquad t' = t + \mathsf{used\text{-}tmps}\ s_{\mathsf{e}} \qquad \mathit{cond} = (\lambda j.\ \mathsf{the}\ (j\ t') = c\ j) \\ \mathit{ret} = (\lambda v_1\ v_2.\ v_1) \qquad \mathit{is} = \mathsf{issue\text{-}expr}\ t\ s_{\mathsf{e}}\ @\ [\mathrm{RMW}\ (a\ j)\ t'\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ (A\ j)\ (L\ j)\ (R\ j)\ (W\ j)] \end{array}}{j \vdash (\mathrm{CAS}\ (\textsc{Tmp}\ (D_{\mathsf{a}},\ a))\ (\textsc{Tmp}\ (D_{\mathsf{c}},\ c))\ s_{\mathsf{e}}\ A\ L\ R\ W,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Skip},\ \mathsf{Suc}\ t'),\ \mathit{is})}$$

$$\frac{j \vdash (s_1,\ t) \rightarrow_{\mathsf{p}} (({s_1}',\ t'),\ \mathit{is})}{j \vdash (\textsc{Seq}\ s_1\ s_2,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Seq}\ {s_1}'\ s_2,\ t'),\ \mathit{is})}$$

$$\frac{}{j \vdash (\textsc{Seq}\ \textsc{Skip}\ s_2,\ t) \rightarrow_{\mathsf{p}} ((s_2,\ t),\ [])}$$

$$\frac{\forall\, \mathit{sop}.\ e \neq \textsc{Tmp}\ \mathit{sop} \qquad e' = \textsc{Tmp}\ (\mathsf{eval\text{-}expr}\ t\ e) \qquad t' = t + \mathsf{used\text{-}tmps}\ e \qquad \mathit{is} = \mathsf{issue\text{-}expr}\ t\ e}{j \vdash (\textsc{Cond}\ e\ s_1\ s_2,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Cond}\ e'\ s_1\ s_2,\ t'),\ \mathit{is})}$$

$$\frac{D \subseteq \mathsf{dom}\ j \qquad \mathsf{isTrue}\ (e\ j)}{j \vdash (\textsc{Cond}\ (\textsc{Tmp}\ (D,\ e))\ s_1\ s_2,\ t) \rightarrow_{\mathsf{p}} ((s_1,\ t),\ [])}$$

$$\frac{D \subseteq \mathsf{dom}\ j \qquad \neg\ \mathsf{isTrue}\ (e\ j)}{j \vdash (\textsc{Cond}\ (\textsc{Tmp}\ (D,\ e))\ s_1\ s_2,\ t) \rightarrow_{\mathsf{p}} ((s_2,\ t),\ [])}$$

$$\frac{}{j \vdash (\textsc{While}\ e\ s,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Cond}\ e\ (\textsc{Seq}\ s\ (\textsc{While}\ e\ s))\ \textsc{Skip},\ t),\ [])}$$

$$\frac{}{j \vdash (\mathrm{S}\textsc{Ghost}\ A\ L\ R\ W,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Skip},\ t),\ [\textsc{Ghost}\ (A\ j)\ (L\ j)\ (R\ j)\ (W\ j)])}$$

$$\frac{}{j \vdash (\mathrm{S}\textsc{Fence},\ t) \rightarrow_{\mathsf{p}} ((\textsc{Skip},\ t),\ [\textsc{Fence}])}$$

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 j be the valuation of temporaries in the current configuration, for every thread configuration $ts_{\mathsf{sb}[i]} = ((s, t), is, j, sb, \mathcal{D}, \mathcal{O})$ where $i < |ts_{\mathsf{sb}}|$ we require: (i) The domain of all intermediate $\textsc{Tmp}$ (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 j 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 sequentially 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:

```
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 ordinarry 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 conlcusion 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 tmps = tmp ⇒ val option
type-synonym sop = tmp set × (tmps ⇒ val) — domain and function

locale valid-sop =
fixes sop :: sop
assumes valid-sop: ⋀D f j.
    ⟦sop=(D,f); D ⊆ dom j⟧
    ⟹
    f j = f (j|‘D)

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

type-synonym lcl = addr set
type-synonym wrt = addr set
type-synonym cond = tmps ⇒ bool
type-synonym ret = val ⇒ val ⇒ val

datatype instr = Read bool addr tmp
| Write bool addr sop acq lcl rel wrt
| RMW addr tmp sop cond ret acq lcl rel wrt
| Fence
| Ghost acq lcl rel wrt

type-synonym instrs = instr list

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

definition owned t = (let (p,instrs,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = t in $\mathcal{O}$)

lemma owned-simp [simp]: owned (p,instrs,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = ($\mathcal{O}$)
⟨*proof*⟩

definition $\mathcal{O}$-sb t = (let (p,instrs,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = t in ($\mathcal{O}$,sb))

lemma $\mathcal{O}$-sb-simp [simp]: $\mathcal{O}$-sb (p,instrs,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = ($\mathcal{O}$,sb)
⟨*proof*⟩

definition released t = (let (p,instrs,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = t in $\mathcal{R}$)

lemma released-simp [simp]: released (p,instrs,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = ($\mathcal{R}$)
⟨*proof*⟩

lemma list-update-id': v = xs ! i ⟹ xs[i := v] = xs
⟨*proof*⟩

lemmas converse-rtranclp-induct5 =
converse-rtranclp-induct [**where** a=(m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) **and** b=(m',sb',$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$), 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
(‹- $\rightarrow_{\mathsf{m}}$ -› [60,60] 100) **and**

storebuffer-step:: (memory $\times$ 'sb $\times$ 'owns $\times$ 'rels $\times$ 'shared) $\Rightarrow$ (memory $\times$ 'sb $\times$ 'owns $\times$ 'rels $\times$ 'shared) $\Rightarrow$ bool (‹- $\rightarrow_{\mathsf{sb}}$ -› [60,60] 100)

locale program =
fixes
program-step :: tmps $\Rightarrow$ 'p $\Rightarrow$ 'p $\times$ instrs $\Rightarrow$ bool (‹-$\vdash$ - $\rightarrow_{\mathsf{p}}$ -› [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 $\times$ 'sb $\times$ 'owns $\times$ 'rels $\times$ 'shared) $\Rightarrow$ (memory $\times$ 'sb $\times$ 'owns $\times$ 'rels $\times$ 'shared) $\Rightarrow$ bool **and**
memop-step ::
(instrs $\times$ tmps $\times$ 'sb $\times$ memory $\times$ 'dirty $\times$ 'owns $\times$ 'rels $\times$ 'shared) $\Rightarrow$
(instrs $\times$ tmps $\times$ 'sb $\times$ memory $\times$ 'dirty $\times$ 'owns $\times$ 'rels $\times$ 'shared) $\Rightarrow$ bool
and
program-step :: tmps $\Rightarrow$ 'p $\Rightarrow$ 'p $\times$ instrs $\Rightarrow$ bool
fixes
record :: 'p $\Rightarrow$ 'p $\Rightarrow$ instrs $\Rightarrow$ 'sb $\Rightarrow$ 'sb
begin

inductive concurrent-step ::
('p,'sb,'dirty,'owns,'rels,'shared) global-config $\Rightarrow$ ('p,'sb,'dirty,'owns,'rels,'shared) global-config $\Rightarrow$ bool
(‹- $\Rightarrow$ -› [60,60] 100)
where
Program:
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
j$\vdash$p $\rightarrow_{\mathsf{p}}$ (p',is') $]\!]$ $\Longrightarrow$
(ts,m,$\mathcal{S}$) $\Rightarrow$ (ts[i:=(p',is@is',j,record p p' is' sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)],m,$\mathcal{S}$)

| Memop:
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
(is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{m}}$ (is',j',sb',m',$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$) $]\!]$
$\Longrightarrow$
(ts,m,$\mathcal{S}$) $\Rightarrow$ (ts[i:=(p,is',j',sb',$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)],m',$\mathcal{S}'$)

| StoreBuffer:
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
(m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$ (m',sb',$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$) $]\!]$ $\Longrightarrow$
(ts,m,$\mathcal{S}$) $\Rightarrow$ (ts[i:=(p,is,j,sb',$\mathcal{D}$,$\mathcal{O}'$,$\mathcal{R}'$)],m',$\mathcal{S}'$)

definition final:: ('p,'sb,'dirty,'owns,'rels,'shared) global-config $\Rightarrow$ bool
where
final c = ($\neg$ ($\exists$c'. c $\Rightarrow$ c'))

lemma store-buffer-steps:
assumes sb-step: storebuffer-step^** $(m,sb,\mathcal{O},\mathcal{R},\mathcal{S})$ $(m',sb',\mathcal{O}',\mathcal{R}',\mathcal{S}')$
shows $\bigwedge$ts. i < length ts $\Longrightarrow$ ts!i = $(p,is,j,sb,\mathcal{D},\mathcal{O},\mathcal{R})$ $\Longrightarrow$
concurrent-step^** $(ts,m,\mathcal{S})$ $(ts[i:=(p,is,j,sb',\mathcal{D},\mathcal{O}',\mathcal{R}')],m',\mathcal{S}')$
⟨*proof*⟩

lemma step-preserves-length-ts:
assumes step: $(ts,m,\mathcal{S}) \Rightarrow (ts',m',\mathcal{S}')$
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 ⇒ addr set ⇒ addr set ⇒ shared (‹- ⊕_ -› [61,1000,60] 61)
where
$\mathcal{S} \oplus_W S \equiv (\lambda a.$ if $a \in S$ then Some $(a \in W)$ else $\mathcal{S}$ a)

definition restrict-shared:: shared ⇒ addr set ⇒ addr set ⇒ shared (‹- ⊖_ -› [51,1000,50] 51)
where
$\mathcal{S} \ominus_A L \equiv (\lambda a.$ if $a \in L$ then None
else (case $\mathcal{S}$ a of None ⇒ None
| Some writeable ⇒ Some $(a \in A \vee$ writeable)))

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

definition shared-le:: shared ⇒ shared ⇒ bool (**infix** ‹$\subseteq_s$› 50)
where
$m_1 \subseteq_s m_2 \equiv m_1 \subseteq_m m_2 \wedge$ read-only $m_1 \subseteq$ read-only m_2

lemma shared-leD: $m_1 \subseteq_s m_2 \Longrightarrow m_1 \subseteq_m m_2 \wedge$ read-only $m_1 \subseteq$ read-only m_2
⟨*proof*⟩

lemma shared-le-map-le: $m_1 \subseteq_s m_2 \Longrightarrow m_1 \subseteq_m m_2$
⟨*proof*⟩

lemma shared-le-read-only-le: $m_1 \subseteq_s m_2 \Longrightarrow$ read-only $m_1 \subseteq$ 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]: $X \cap - L = X - L$
⟨*proof*⟩

lemma dom-restrict-shared [simp]: $\mathrm{dom}\ (m \ominus_{\mathsf{A}} L) = \mathrm{dom}\ m - L$
⟨*proof*⟩

lemma restrict-shared-UNIV [simp]: $(m \ominus_{\mathsf{A}} \mathrm{UNIV}) = \mathrm{Map.empty}$
⟨*proof*⟩

lemma restrict-shared-empty [simp]: $(\mathrm{Map.empty} \ominus_{\mathsf{A}} L) = \mathrm{Map.empty}$
⟨*proof*⟩

lemma restrict-shared-in [simp]: $a \in L \Longrightarrow (m \ominus_{\mathsf{A}} L)\ a = \mathrm{None}$
⟨*proof*⟩

lemma restrict-shared-out: $a \notin L \Longrightarrow (m \ominus_{\mathsf{A}} L)\ a =$
$\text{map-option}\ (\lambda\text{writeable}.\ (a \in A \lor \text{writeable}))\ (m\ a)$
⟨*proof*⟩

lemma restrict-shared-out′[simp]:
$a \notin L \Longrightarrow m\ a = \mathrm{Some}\ \text{writeable} \Longrightarrow (m \ominus_{\mathsf{A}} L)\ a = \mathrm{Some}\ (a \in A \lor \text{writeable})$
⟨*proof*⟩

lemma augment-mono-map′: $A \subseteq_{\mathsf{m}} B \Longrightarrow (A \oplus_{\mathsf{x}} C) \subseteq_{\mathsf{m}} (B \oplus_{\mathsf{x}} C)$
⟨*proof*⟩

lemma augment-mono-map: $A \subseteq_{\mathsf{s}} B \Longrightarrow (A \oplus_{\mathsf{x}} C) \subseteq_{\mathsf{s}} (B \oplus_{\mathsf{x}} C)$
⟨*proof*⟩

lemma restrict-mono-map: $A \subseteq_{\mathsf{s}} B \Longrightarrow (A \ominus_{\mathsf{x}} C) \subseteq_{\mathsf{s}} (B \ominus_{\mathsf{x}} C)$
⟨*proof*⟩

lemma augment-mono-aux: $\mathrm{dom}\ A \subseteq \mathrm{dom}\ B \Longrightarrow \mathrm{dom}\ (A \oplus_{\mathsf{x}} C) \subseteq \mathrm{dom}\ (B \oplus_{\mathsf{x}} C)$
⟨*proof*⟩

lemma restrict-mono-aux: $\mathrm{dom}\ A \subseteq \mathrm{dom}\ B \Longrightarrow \mathrm{dom}\ (A \ominus_{\mathsf{x}} C) \subseteq \mathrm{dom}\ (B \ominus_{\mathsf{x}} C)$
⟨*proof*⟩

lemma read-only-mono: $S \subseteq_{\mathsf{m}} S' \Longrightarrow a \in \text{read-only}\ S \Longrightarrow a \in \text{read-only}\ S'$
⟨*proof*⟩

lemma in-read-only-restrict-conv:
$a \in \text{read-only}\ (\mathcal{S} \ominus_{\mathsf{A}} L) = (a \in \text{read-only}\ \mathcal{S} \land a \notin L \land a \notin A)$
⟨*proof*⟩

lemma in-read-only-augment-conv: a $\in$ read-only ($\mathcal{S} \oplus_{\mathsf{W}}$ R) = (if a $\in$ R then a $\notin$ W else a $\in$ read-only $\mathcal{S}$)
⟨*proof*⟩

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

lemma read-only-dom: read-only $\mathcal{S} \subseteq$ dom $\mathcal{S}$
⟨*proof*⟩

lemma read-only-empty [simp]: read-only Map.empty = {}
⟨*proof*⟩

lemma restrict-shared-fuse: S $\ominus_{\mathsf{A}}$ L $\ominus_{\mathsf{B}}$ M = (S $\ominus_{(\mathsf{A}\ \cup\ \mathsf{B})}$ (L $\cup$ M))
⟨*proof*⟩

lemma restrict-shared-empty-set [simp]: S $\ominus_{\{\}}$ {} = S
⟨*proof*⟩

definition augment-rels:: addr set $\Rightarrow$ addr set $\Rightarrow$ rels $\Rightarrow$ rels
where
augment-rels S R $\mathcal{R}$ = (λa. if a $\in$ R
then (case $\mathcal{R}$ a of
None $\Rightarrow$ Some (a $\in$ S)
| Some s $\Rightarrow$ Some (s $\wedge$ (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 $\times$ tmps $\times$ unit $\times$ memory $\times$ bool $\times$ owns $\times$ ′rels $\times$ shared) $\Rightarrow$
(instrs $\times$ tmps $\times$ unit $\times$ memory $\times$ bool $\times$ owns $\times$ ′rels $\times$ shared) $\Rightarrow$ bool
(‹- $\rightarrow$ -› [60,60] 100)
where
Read: (Read volatile a t # is,j, x, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$
(is, j (t$\mapsto$m a), x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| WriteNonVolatile:
(Write False a (D,f) A L R W#is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$
(is, j, x, m(a := f j), $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| WriteVolatile:
(Write True a (D,f) A L R W# is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$
(is, j, x, m(a:=f j), True, $\mathcal{O} \cup$ A $-$ R, emp, $\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)

| Fence:

(Fence # is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$ (is, j,x, m, False, $\mathcal{O}$, emp, $\mathcal{S}$)

| RMWReadOnly:
⟦¬ cond (j(t↦m a))⟧ $\Longrightarrow$
(RMW a t (D,f) cond ret A L R W # is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$ (is, j(t↦m a),x,m, False, $\mathcal{O}$, emp, $\mathcal{S}$)

| RMWWrite:
⟦cond (j(t↦m a))⟧ $\Longrightarrow$
(RMW a t (D,f) cond ret A L R W# is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$
(is, j(t↦ret (m a) (f(j(t↦m a)))),x, m(a:= f(j(t↦m a))), False,$\mathcal{O}$ $\cup$ A $-$ R, emp, $\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)

| Ghost:
(Ghost A L R W # is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow$
(is, j, x, m, $\mathcal{D}$, $\mathcal{O}$ $\cup$ A $-$ R, aug (dom $\mathcal{S}$) R $\mathcal{R}$, $\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ 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 × tmps × unit × memory × bool × owns × rels × shared) ⇒
(instrs × tmps × unit × memory × bool × owns × rels × shared) ⇒ bool
(‹- → -› [60,60] 100)
where
direct-memop-step ≡ direct-memop-step.gen-direct-memop-step

term x → Y

abbreviation direct-memop-steps ::
(instrs × tmps × unit × memory × bool × owns × rels × shared) ⇒
(instrs × tmps × unit × memory × bool × owns × rels × shared)
⇒ bool
(‹- $\rightarrow^*$ -› [60,60] 100)
where
direct-memop-steps == (direct-memop-step)^**

term x $\rightarrow^*$ Y

interpretation virtual-memop-step: gen-direct-memop-step () (λS R $\mathcal{R}$. ()) ⟨*proof*⟩

abbreviation virtual-memop-step :: (instrs × tmps × unit × memory × bool × owns × unit × shared) ⇒
(instrs × tmps × unit × memory × bool × owns × unit × shared) ⇒ bool
(‹- $\rightarrow_{\mathsf{v}}$ -› [60,60] 100)
where
virtual-memop-step ≡ virtual-memop-step.gen-direct-memop-step

term x $\rightarrow_{\mathsf{v}}$ Y

abbreviation virtual-memop-steps ::
(instrs × tmps × unit × memory × bool × owns × unit × shared) ⇒
(instrs × tmps × unit × memory × bool × owns × unit × shared)
⇒ bool
(‹- $\rightarrow_{\mathsf{v}}{}^*$ -› [60,60] 100)
where
virtual-memop-steps == (virtual-memop-step)^**

term x →* Y

lemma virtual-memop-step-simulates-direct-memop-step:
assumes step:
(is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) → (is′, j′, x′, m′, $\mathcal{D}'$, $\mathcal{O}'$, $\mathcal{R}'$, $\mathcal{S}'$)
shows (is, j, x, m, $\mathcal{D}$, $\mathcal{O}$, (), $\mathcal{S}$) $\rightarrow_{\mathsf{v}}$ (is′, j′, x′, m′, $\mathcal{D}'$, $\mathcal{O}'$, (), $\mathcal{S}'$)
⟨*proof*⟩

A.4 Safe Configurations of Virtual Machines

inductive safe-direct-memop-state :: owns list ⇒ nat ⇒
(instrs × tmps × memory × bool × owns × shared) ⇒ bool
(‹-,-⊢ - √› [60,60,60] 100)
where
Read: ⟦a ∈ $\mathcal{O}$ ∨ a ∈ read-only $\mathcal{S}$ ∨ (volatile ∧ a ∈ dom $\mathcal{S}$);
volatile ⟶ ¬ $\mathcal{D}$ ⟧
⟹
$\mathcal{O}$s,i⊢(Read volatile a t # is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

| WriteNonVolatile:
⟦a ∈ $\mathcal{O}$; a ∉ dom $\mathcal{S}$⟧
⟹
$\mathcal{O}$s,i⊢(Write False a (D,f) A L R W#is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

| WriteVolatile:
⟦∀ j < length $\mathcal{O}$s. i≠j ⟶ a ∉ $\mathcal{O}$s!j;
A ⊆ dom $\mathcal{S}$ ∪ $\mathcal{O}$; L ⊆ A; R ⊆ $\mathcal{O}$; A ∩ R = {};
∀ j < length $\mathcal{O}$s. i≠j ⟶ A ∩ $\mathcal{O}$s!j = {};
a ∉ read-only $\mathcal{S}$⟧
⟹
$\mathcal{O}$s,i⊢(Write True a (D,f) A L R W# is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

| Fence:
$\mathcal{O}$s,i⊢(Fence # is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

| Ghost:
⟦A ⊆ dom $\mathcal{S}$ ∪ $\mathcal{O}$; L ⊆ A; R ⊆ $\mathcal{O}$; A ∩ R = {};
∀ j < length $\mathcal{O}$s. i≠j ⟶ A ∩ $\mathcal{O}$s!j = {}⟧

⟹
𝒪s,i⊢(Ghost A L R W# is, j, m, 𝒟, 𝒪, 𝒮)√

| RMWReadOnly:
⟦¬ cond (j(t↦m a)); a ∈ 𝒪 ∨ a ∈ dom 𝒮⟧ ⟹
𝒪s,i⊢(RMW a t (D,f) cond ret A L R W# is, j, m, 𝒟, 𝒪, 𝒮)√

| RMWWrite:
⟦cond (j(t↦m a));
∀j < length 𝒪s. i≠j ⟶ a ∉ 𝒪s!j;
A ⊆ dom 𝒮 ∪ 𝒪; L ⊆ A; R ⊆ 𝒪; A ∩ R = {};
∀j < length 𝒪s. i≠j ⟶ A ∩ 𝒪s!j = {};
a ∉ read-only 𝒮⟧
⟹
𝒪s,i⊢(RMW a t (D,f) cond ret A L R W# is, j, m, 𝒟, 𝒪, 𝒮)√

| Nil: 𝒪s,i⊢([], j, m, 𝒟, 𝒪, 𝒮)√

inductive safe-delayed-direct-memop-state :: owns list ⇒ rels list ⇒ nat ⇒
(instrs × tmps × memory × bool × owns × shared) ⇒ bool
(‹-,-,-⊢ - √› [60,60,60,60] 100)
where
Read: ⟦a ∈ 𝒪 ∨ a ∈ read-only 𝒮 ∨ (volatile ∧ a ∈ dom 𝒮);
∀j < length 𝒪s. i≠j ⟶ (ℛs!j) a ≠ Some False; — no release of unshared address
¬ volatile ⟶ (∀j < length 𝒪s. i≠j ⟶ a ∉ dom (ℛs!j));
volatile ⟶ ¬ 𝒟 ⟧
⟹
𝒪s,ℛs,i⊢(Read volatile a t # is, j, m, 𝒟, 𝒪, 𝒮)√

| WriteNonVolatile:
⟦a ∈ 𝒪; a ∉ dom 𝒮; ∀j < length 𝒪s. i≠j ⟶ a ∉ dom (ℛs!j)⟧
⟹
𝒪s,ℛs,i⊢(Write False a (D,f) A L R W#is, j, m, 𝒟, 𝒪, 𝒮)√

| WriteVolatile:
⟦∀j < length 𝒪s. i≠j ⟶ a ∉ (𝒪s!j ∪ dom (ℛs!j));
A ⊆ dom 𝒮 ∪ 𝒪; L ⊆ A; R ⊆ 𝒪; A ∩ R = {};
∀j < length 𝒪s. i≠j ⟶ A ∩ (𝒪s!j ∪ dom (ℛs!j)) = {};
a ∉ read-only 𝒮⟧
⟹
𝒪s,ℛs,i⊢(Write True a (D,f) A L R W# is, j, m, 𝒟, 𝒪, 𝒮)√

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

| Ghost:
⟦A ⊆ dom 𝒮 ∪ 𝒪; L ⊆ A; R ⊆ 𝒪; A ∩ R = {};
∀j < length 𝒪s. i≠j ⟶ A ∩ (𝒪s!j ∪ dom (ℛs!j)) = {}⟧
⟹
𝒪s,ℛs,i⊢(Ghost A L R W# is, j, m, 𝒟, 𝒪, 𝒮)√

| RMWReadOnly:
  ⟦¬ cond (j(t↦m a)); a ∈ $\mathcal{O}$ ∨ a ∈ dom $\mathcal{S}$;
  ∀ j < length $\mathcal{O}$s. i≠j ⟶ ($\mathcal{R}$s!j) a ≠ Some False — no release of unshared address⟧
  ⟹
  $\mathcal{O}$s,$\mathcal{R}$s,i⊢(RMW a t (D,f) cond ret A L R W# is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

| RMWWrite:
  ⟦cond (j(t↦m a)); a ∈ $\mathcal{O}$ ∨ a ∈ dom $\mathcal{S}$;
   ∀ j < length $\mathcal{O}$s. i≠j ⟶ a ∉ ($\mathcal{O}$s!j ∪ dom ($\mathcal{R}$s!j));
   A ⊆ dom $\mathcal{S}$ ∪ $\mathcal{O}$; L ⊆ A; R ⊆ $\mathcal{O}$; A ∩ R = {};
   ∀ j < length $\mathcal{O}$s. i≠j ⟶ A ∩ ($\mathcal{O}$s!j ∪ dom ($\mathcal{R}$s!j)) = {};
   a ∉ read-only $\mathcal{S}$⟧
  ⟹
  $\mathcal{O}$s,$\mathcal{R}$s,i⊢(RMW a t (D,f) cond ret A L R W# is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

| Nil: $\mathcal{O}$s,$\mathcal{R}$s,i⊢([], j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√

lemma memop-safe-delayed-implies-safe-free-flowing:
  assumes safe-delayed: $\mathcal{O}$s,$\mathcal{R}$s,i⊢(is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√
  shows $\mathcal{O}$s,i⊢(is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√
⟨*proof*⟩

lemma memop-empty-rels-safe-free-flowing-implies-safe-delayed:
  assumes safe: $\mathcal{O}$s,i⊢(is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√
  assumes empty: ∀ $\mathcal{R}$ ∈ set $\mathcal{R}$s. $\mathcal{R}$ = Map.empty
  assumes leq: length $\mathcal{O}$s = length $\mathcal{R}$s
  assumes unowned-shared: (∀ a. (∀ i < length $\mathcal{O}$s. a ∉ ($\mathcal{O}$s!i)) ⟶ a ∈ dom $\mathcal{S}$)
  assumes Os-i: $\mathcal{O}$s!i = $\mathcal{O}$
  shows $\mathcal{O}$s,$\mathcal{R}$s,i⊢(is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)√
⟨*proof*⟩

inductive id-storebuffer-step::
  (memory × unit × owns × rels × shared) ⇒ (memory × unit × owns × rels × shared) ⇒ bool (‹- $\rightarrow_{\mathsf{I}}$ -› [60,60] 100)
where
  Id: (m,x,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{I}}$ (m,x,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$)

definition empty-storebuffer-step:: (memory × 'sb × 'owns × 'rels × 'shared) ⇒ (memory × 'sb × 'owns × 'rels × 'shared) ⇒ bool
where
empty-storebuffer-step c c' = False

context program
begin

abbreviation direct-concurrent-step ::
  ('p,unit,bool,owns,rels,shared) global-config ⇒ ('p,unit,bool,owns,rels,shared) global-config ⇒ bool

(‹- $\Rightarrow_d$ -› [100,60] 100)
where
direct-concurrent-step ≡
computation.concurrent-step direct-memop-step.gen-direct-memop-step empty-storebuffer-step program-step
(λp p′ is sb. sb)

abbreviation direct-concurrent-steps::
('p,unit,bool,owns,rels,shared) global-config ⇒ ('p,unit,bool,owns,rels,shared) global-config ⇒ bool
(‹- $\Rightarrow_d{}^*$ -› [60,60] 100)
where
direct-concurrent-steps == direct-concurrent-step^**

abbreviation virtual-concurrent-step ::
('p,unit,bool,owns,unit,shared) global-config ⇒ ('p,unit,bool,owns,unit,shared) global-config ⇒ bool
(‹- $\Rightarrow_v$ -› [100,60] 100)
where
virtual-concurrent-step ≡
computation.concurrent-step virtual-memop-step.gen-direct-memop-step empty-storebuffer-step program-step
(λp p′ is sb. sb)

abbreviation virtual-concurrent-steps::
('p,unit,bool,owns,unit,shared) global-config ⇒ ('p,unit,bool,owns,unit,shared) global-config ⇒ bool
(‹- $\Rightarrow_v{}^*$ -› [60,60] 100)
where
virtual-concurrent-steps == virtual-concurrent-step^**

term x $\Rightarrow_v$ Y
term x $\Rightarrow_d$ Y

term x $\Rightarrow_d{}^*$ Y
term x $\Rightarrow_v{}^*$ Y

end

definition
safe-reach step safe cfg ≡
∀ cfg′. step^** cfg cfg′ ⟶ safe cfg′

lemma safe-reach-safe-refl: safe-reach step safe cfg ⟹ safe cfg
⟨*proof*⟩

lemma safe-reach-safe-rtrancl: safe-reach step safe cfg ⟹ step^** cfg cfg′ ⟹ safe cfg′

⟨*proof*⟩

lemma safe-reach-steps: safe-reach step safe cfg $\Longrightarrow$ step^** cfg cfg′ $\Longrightarrow$ safe-reach step safe cfg′
⟨*proof*⟩

lemma safe-reach-step: safe-reach step safe cfg $\Longrightarrow$ step cfg cfg′ $\Longrightarrow$ 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 $\Rightarrow_{\mathsf{d}}{}^{*}$ 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 $\Rightarrow_{\mathsf{v}}{}^{*}$ cfg′ ⟶ safe cfg′
⟨*proof*⟩
end

definition
safe-free-flowing cfg ≡ let (ts,m,$\mathcal{S}$) = cfg
in (∀ i < length ts. let (p,is,j,x,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = ts!i in
map owned ts,i ⊢(is,j,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{S}$)$\surd$)

lemma safeE: ⟦safe-free-flowing (ts,m,$\mathcal{S}$);i<length ts; ts!i=(p,is,j,x,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
$\Longrightarrow$ map owned ts,i ⊢(is,j,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{S}$)$\surd$
⟨*proof*⟩

definition
safe-delayed cfg ≡ let (ts,m,$\mathcal{S}$) = cfg
in (∀ i < length ts. let (p,is,j,x,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = ts!i in
map owned ts,map released ts,i ⊢(is,j,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{S}$)$\surd$)

lemma safe-delayedE: ⟦safe-delayed (ts,m,$\mathcal{S}$);i<length ts; ts!i=(p,is,j,x,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
$\Longrightarrow$ map owned ts,map released ts,i ⊢(is,j,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{S}$)$\surd$
⟨*proof*⟩

definition remove-rels ≡ map (λ(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$). (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,()))

theorem (**in** program) virtual-simulates-direct-step:

assumes step: (ts,m,$\mathcal{S}$) $\Rightarrow_{\mathsf{d}}$ (ts′,m′,$\mathcal{S}$′)
shows (remove-rels ts,m,$\mathcal{S}$) $\Rightarrow_{\mathsf{v}}$ (remove-rels ts′,m′,$\mathcal{S}$′)
⟨*proof*⟩

lemmas converse-rtranclp-induct-sbh-steps = converse-rtranclp-induct
[of - (ts,m,$\mathcal{S}$) (ts′,m′,$\mathcal{S}$′), split-rule,
consumes 1, case-names refl step]

theorem (**in** program) virtual-simulates-direct-steps:
assumes steps: (ts,m,$\mathcal{S}$) $\Rightarrow_{\mathsf{d}}^*$ (ts′,m′,$\mathcal{S}$′)
shows (remove-rels ts,m,$\mathcal{S}$) $\Rightarrow_{\mathsf{v}}^*$ (remove-rels ts′,m′,$\mathcal{S}$′)
⟨*proof*⟩

locale simple-ownership-distinct =
fixes ts::(′p,′sb,′dirty,owns,′rels) thread-config list
assumes simple-ownership-distinct:
$\bigwedge$i j p_i is_i $\mathcal{O}_i$ $\mathcal{R}_i$ $\mathcal{D}_i$ j_i sb_i p_j is_j $\mathcal{O}_j$ $\mathcal{R}_j$ $\mathcal{D}_j$ j_j sb_j.
⟦i < length ts; j < length ts; i ≠ j;
ts!i = (p_i,is_i,j_i,sb_i,$\mathcal{D}_i$,$\mathcal{O}_i$,$\mathcal{R}_i$); ts!j = (p_j,is_j,j_j,sb_j,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$)
⟧ $\Longrightarrow$ $\mathcal{O}_i \cap \mathcal{O}_j$ = {}

lemma (**in** simple-ownership-distinct)
simple-ownership-distinct-nth-update:
$\bigwedge$i p is j $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ xs sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
∀ j < length ts. i≠j ⟶ (let (p_j,is_j,j_j,sb_j,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$) = ts!j
in ($\mathcal{O}$′) ∩ ($\mathcal{O}_j$) ={}) ⟧ $\Longrightarrow$
simple-ownership-distinct (ts[i := (p′,is′,j′,sb′,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′)])
⟨*proof*⟩

locale read-only-unowned =
fixes $\mathcal{S}$::shared **and** ts::(′p,′sb,′dirty,owns,′rels) thread-config list
assumes read-only-unowned:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟧
$\Longrightarrow$
$\mathcal{O}$ ∩ read-only $\mathcal{S}$ = {}

lemma (**in** read-only-unowned)
read-only-unowned-nth-update:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ acq j sb.
⟦i < length ts; $\mathcal{O}$ ∩ read-only $\mathcal{S}$ = {}⟧ $\Longrightarrow$
read-only-unowned $\mathcal{S}$ (ts[i := (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

locale unowned-shared =
fixes $\mathcal{S}$::shared **and** ts::(′p,′sb,′dirty,owns,′rels) thread-config list
assumes unowned-shared: − $\bigcup$ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ‘ set ts) ⊆ 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,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes subset: $\mathcal{O} \subseteq \mathcal{O}'$
shows unowned-shared $\mathcal{S}$ (ts[i := (p′,is′,xs′,sb′,$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)])
⟨*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,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-unowned-others:
$\forall$ j<length (map owned ts). i $\neq$ j $\longrightarrow$
(let $\mathcal{O}_j$ = (map owned ts)!j in a $\notin$ $\mathcal{O}_j$)

shows a $\in$ $\mathcal{O}$ $\vee$ a $\in$ dom $\mathcal{S}$
⟨*proof*⟩

lemma (**in** unowned-shared) unowned-shared′:
assumes notin: $\forall$ i < length ts. a $\notin$ owned (ts!i)
shows a $\in$ dom $\mathcal{S}$
⟨*proof*⟩

lemma unowned-shared-def′: unowned-shared $\mathcal{S}$ ts = ($\forall$ a. ($\forall$ i < length ts. a $\notin$ owned (ts!i)) $\longrightarrow$ a $\in$ dom $\mathcal{S}$)
⟨*proof*⟩

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

lemma initial-empty-rels: initial (ts,m,$\mathcal{S}$) $\Longrightarrow$ $\forall$ $\mathcal{R}$ $\in$ set (map released ts). $\mathcal{R}$ = Map.empty
⟨*proof*⟩

lemma initial-unowned-shared: initial (ts,m,$\mathcal{S}$) $\Longrightarrow$ unowned-shared $\mathcal{S}$ 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: j$\vdash$ p $\rightarrow_{\mathsf{p}}$ (p′,is′) $\Longrightarrow$ p′ $\neq$ p $\vee$ is′ $\neq$ []

The assumption 'progress' could be avoided if we introduce stuttering steps in lemma undo-local-step or make the scheduling of threads explicit, such that we can directly express that 'thread i does not make a step'.**lemma** (**in** program-progress) undo-local-step:
assumes step: $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{d}} (ts',m',\mathcal{S}')$
assumes i-bound: $i < \text{length } ts$
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: $\forall j < \text{length } ts.\ j \neq i \longrightarrow u\text{-}ts!j = ts!j$
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: $\forall a.\ a \notin u\text{-}owns \longrightarrow u\text{-}m\ a = m\ a$
assumes u-m-shared: $\forall a.\ a \in u\text{-}owns \longrightarrow a \in \text{dom } u\text{-}shared \longrightarrow u\text{-}m\ a = m\ a$
assumes u-shared: $\forall a.\ a \notin u\text{-}owns \longrightarrow a \notin \text{owned } (ts!i) \longrightarrow u\text{-}shared\ a = \mathcal{S}\ a$
assumes dist: simple-ownership-distinct u-ts
assumes dist-ts: simple-ownership-distinct ts
shows $\exists u\text{-}ts'\ u\text{-}shared'\ u\text{-}m'.\ (u\text{-}ts,u\text{-}m,u\text{-}shared) \Rightarrow_{\mathsf{d}} (u\text{-}ts',u\text{-}m',u\text{-}shared') \wedge$
— thread i is unchanged
$u\text{-}ts'!i = u\text{-}ts!i \wedge$
$(\forall a \in u\text{-}owns.\ u\text{-}shared'\ a = u\text{-}shared\ a) \wedge$
$(\forall a \in u\text{-}owns.\ \mathcal{S}'\ a = \mathcal{S}\ a) \wedge$
$(\forall a \in u\text{-}owns.\ u\text{-}m'\ a = u\text{-}m\ a) \wedge$
$(\forall a \in u\text{-}owns.\ m'\ a = m\ a) \wedge$

— other threads are simulated
$(\forall j < \text{length } ts.\ j \neq i \longrightarrow u\text{-}ts'!j = ts'!j) \wedge$
$(\forall a.\ a \notin u\text{-}owns \longrightarrow a \notin \text{owned } (ts!i) \longrightarrow u\text{-}shared'\ a = \mathcal{S}'\ a) \wedge$
$(\forall a.\ a \notin u\text{-}owns \longrightarrow u\text{-}m'\ a = m'\ a)$
⟨*proof*⟩

theorem (**in** program) safe-step-preserves-simple-ownership-distinct:
assumes step: $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{d}} (ts',m',\mathcal{S}')$
assumes safe: safe-delayed $(ts,m,\mathcal{S})$
assumes dist: simple-ownership-distinct ts
shows simple-ownership-distinct ts′
⟨*proof*⟩

theorem (**in** program) safe-step-preserves-read-only-unowned:
assumes step: $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{d}} (ts',m',\mathcal{S}')$
assumes safe: safe-delayed $(ts,m,\mathcal{S})$
assumes dist: simple-ownership-distinct ts
assumes ro-unowned: read-only-unowned $\mathcal{S}$ ts
shows read-only-unowned $\mathcal{S}'$ ts′
⟨*proof*⟩

theorem (**in** program) safe-step-preserves-unowned-shared:
assumes step: $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{d}} (ts',m',\mathcal{S}')$
assumes safe: safe-delayed $(ts,m,\mathcal{S})$

assumes dist: simple-ownership-distinct ts
assumes unowned-shared: unowned-shared $\mathcal{S}$ ts
shows unowned-shared $\mathcal{S}'$ 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: $\bigwedge$l. l < k $\Longrightarrow$ c (n+l) $\Rightarrow_d$ c (n + (Suc l))

abbreviation (**in** program)
trace ≡ program-trace program-step

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

lemma split-less-Suc: (∀x<Suc k. P x) = (P k ∧ (∀x<k. P x))
⟨*proof*⟩

lemma split-le-Suc: (∀x≤Suc k. P x) = (P (Suc k) ∧ (∀x≤k. P x))
⟨*proof*⟩

lemma (**in** program) steps-to-trace:
assumes steps: x $\Rightarrow_d{}^*$ y
shows ∃c k. trace c 0 k ∧ c 0 = x ∧ c k = y
⟨*proof*⟩

lemma (**in** program) trace-preserves-length-ts:
$\bigwedge$l x. trace c n k $\Longrightarrow$ l ≤ k $\Longrightarrow$ x ≤ k $\Longrightarrow$ length (fst (c (n + l))) = length (fst (c (n + x)))
⟨*proof*⟩

lemma (**in** program) trace-preserves-simple-ownership-distinct:
assumes dist: simple-ownership-distinct (fst (c n))
shows $\bigwedge$l. trace c n k $\Longrightarrow$ (∀x < k. safe-delayed (c (n + x))) $\Longrightarrow$
l ≤ k $\Longrightarrow$ simple-ownership-distinct (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 $\bigwedge$l. trace c n k $\Longrightarrow$ (∀x < k. safe-delayed (c (n + x))) $\Longrightarrow$
l ≤ k $\Longrightarrow$ read-only-unowned (snd (snd (c (n + l)))) (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 $\bigwedge$l. trace c n k $\Longrightarrow$ ($\forall$x < k. safe-delayed (c (n + x))) $\Longrightarrow$
l $\leq$ k $\Longrightarrow$ unowned-shared (snd (snd (c (n + l)))) (fst (c (n + l)))
⟨*proof*⟩

theorem (**in** program-progress) undo-local-steps:
assumes steps: trace c n k
assumes c-n: c n = (ts,m,$\mathcal{S}$)
assumes unchanged: $\forall$l $\leq$ k. ($\forall$ts$_l$ $\mathcal{S}_l$ m$_l$. c (n + l) = (ts$_l$,m$_l$,$\mathcal{S}_l$) $\longrightarrow$ ts$_l$!i=ts!i)
assumes safe: safe-delayed (u-ts, u-m, u-shared)
assumes leq: length u-ts = length ts
assumes i-bound: i < length ts
assumes others-same: $\forall$j < length ts. j$\neq$i $\longrightarrow$ u-ts!j = ts!j
assumes u-ts-i: u-ts!i=(u-p,u-is,u-tmps,u-sb,u-dirty,u-owns,u-rels)
assumes u-m-other: $\forall$a. a $\notin$ u-owns $\longrightarrow$ u-m a = m a
assumes u-m-shared: $\forall$a. a $\in$ u-owns $\longrightarrow$ a $\in$ dom u-shared $\longrightarrow$ u-m a = m a
assumes u-shared: $\forall$a. a $\notin$ u-owns $\longrightarrow$ a $\notin$ owned (ts!i) $\longrightarrow$ u-shared a = $\mathcal{S}$ a
assumes dist: simple-ownership-distinct u-ts
assumes dist-ts: simple-ownership-distinct ts
assumes safe-orig: $\forall$x. x < k $\longrightarrow$ safe-delayed (c (n + x))
shows $\exists$c$'$ l. l $\leq$ k $\wedge$ 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))) $\wedge$
(l < k $\longrightarrow$ $\neg$ safe-delayed (c$'$ (n + l))) $\wedge$

($\forall$x $\leq$ l. $\forall$ts$_x$ $\mathcal{S}_x$ m$_x$ ts$_x'$ $\mathcal{S}_x'$ m$_x'$. c (n + x) = (ts$_x$,m$_x$,$\mathcal{S}_x$) $\longrightarrow$ c$'$ (n+ x) = (ts$_x'$,m$_x'$,$\mathcal{S}_x'$) $\longrightarrow$
ts$_x'$!i=u-ts!i $\wedge$
($\forall$a $\in$ u-owns. $\mathcal{S}_x'$ a = u-shared a) $\wedge$
($\forall$a $\in$ u-owns. $\mathcal{S}_x$ a = $\mathcal{S}$ 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$_x$ $\mathcal{S}_x$ m$_x$ ts$_x'$ $\mathcal{S}_x'$ m$_x'$. c (n + x) = (ts$_x$,m$_x$,$\mathcal{S}_x$) $\longrightarrow$ c$'$ (n + x) = (ts$_x'$,m$_x'$,$\mathcal{S}_x'$) $\longrightarrow$
($\forall$j < length ts$_x$. j$\neq$i $\longrightarrow$ ts$_x'$!j = ts$_x$!j) $\wedge$
($\forall$a. a $\notin$ u-owns $\longrightarrow$ a $\notin$ owned (ts!i) $\longrightarrow$ $\mathcal{S}_x'$ a = $\mathcal{S}_x$ a) $\wedge$
($\forall$a. a $\notin$ u-owns $\longrightarrow$ m$_x'$ a = m$_x$ a))

⟨*proof*⟩

locale program-safe-reach-upto = program +
fixes n **fixes** safe **fixes** c$_0$
assumes safe-config: ⟦k $\leq$ n; trace c 0 k; c 0 = c$_0$; l $\leq$ k ⟧ $\Longrightarrow$ 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 $\leq$ 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
  ($\forall$ l $\leq$ k. fst (c l)!i = fst (c k)!i) $\vee$
  — thread i has a last step in the trace
  ($\exists$ last < k.
    fst (c last)!i $\neq$ fst (c (Suc last))!i $\wedge$
    ($\forall$ l. last < l $\longrightarrow$ l $\leq$ k $\longrightarrow$ 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 $\leq$ 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 =
   $\mathsf{Write_{sb}}$ bool addr sop val acq lcl rel wrt
 | $\mathsf{Read_{sb}}$ bool addr tmp val
 | $\mathsf{Prog_{sb}}$ ′p ′p instrs
 | $\mathsf{Ghost_{sb}}$ 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
  (‹- $\rightarrow_{\mathsf{f}}$ -› [60,60] 100)
where
  $\mathsf{Write_{sb}}$: ⟦$\mathcal{O}'$ = (if volatile then $\mathcal{O}$ ∪ A − R else $\mathcal{O}$);

$\mathcal{S}'$ = (if volatile then $\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L else $\mathcal{S}$);
$\mathcal{R}'$=(if volatile then Map.empty else $\mathcal{R}$)⟧
$\Longrightarrow$
(m, $\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W# rs,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m(a := v), rs,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
| $\text{Read}_{\mathsf{sb}}$: (m, $\text{Read}_{\mathsf{sb}}$ volatile a t v#rs,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m, rs,$\mathcal{O}$,$\mathcal{R}$, $\mathcal{S}$)
| $\text{Prog}_{\mathsf{sb}}$: (m, $\text{Prog}_{\mathsf{sb}}$ p p′ is#rs,$\mathcal{O}$,$\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m, rs,$\mathcal{O}$,$\mathcal{R}$, $\mathcal{S}$)
| Ghost: (m, $\text{Ghost}_{\mathsf{sb}}$ A L R W# rs,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m, rs,$\mathcal{O} \cup$ A $-$ R, augment-rels (dom $\mathcal{S}$) R $\mathcal{R}$, $\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)

abbreviation flush-steps::memory × ′p store-buffer × owns × rels × shared ⇒ memory × ′p store-buffer × owns × rels × shared⇒ bool
(‹- $\rightarrow_{\mathsf{f}}^*$ -› [60,60] 100)
where
flush-steps == flush-step^**

term x $\rightarrow_{\mathsf{f}}^*$ Y

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

inductive store-buffer-step:: memory × ′p store-buffer × ′owns × ′rels × ′shared ⇒ memory × ′p memref list × ′owns × ′rels × ′shared ⇒ bool
(‹- $\rightarrow_{\mathsf{w}}$ -› [60,60] 100)
where
$\text{SBWrite}_{\mathsf{sb}}$:
(m, $\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W# rs,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{w}}$ (m(a := v), rs,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$)

abbreviation store-buffer-steps::memory × ′p store-buffer × ′owns × ′rels × ′shared ⇒ memory × ′p store-buffer × ′owns × ′rels × ′shared⇒ bool
(‹- →w* -› [60,60] 100)
where
store-buffer-steps == store-buffer-step^**

term x →w* Y

fun buffered-val :: ′p memref list ⇒ addr ⇒ val option
where
buffered-val [] a = None
| buffered-val (r # rs) a′ =
(case r of
$\text{Write}_{\mathsf{sb}}$ volatile a - v - - - - ⇒ (case buffered-val rs a′ of
None ⇒ (if a′=a then Some v else None)
| Some v′ ⇒ Some v′)
| - ⇒ buffered-val rs a′)

definition address-of :: ′p memref ⇒ addr set
where
address-of r = (case r of $\text{Write}_{\mathsf{sb}}$ volatile a - v - - - - ⇒ {a} | $\text{Read}_{\mathsf{sb}}$ volatile a t v ⇒ {a} |
- ⇒ {})

lemma address-of-simps [simp]:
address-of (Write$_{sb}$ volatile a sop v A L R W) = {a}
address-of (Read$_{sb}$ volatile a t v) = {a}
address-of (Prog$_{sb}$ p p′ is) = {}
address-of (Ghost$_{sb}$ A L R W) = {}
⟨*proof*⟩

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

lemma is-volatile-simps [simp]:
is-volatile (Write$_{sb}$ volatile a sop v A L R W) = volatile
is-volatile (Read$_{sb}$ volatile a t v) = volatile
is-volatile (Prog$_{sb}$ p p′ is) = False
is-volatile (Ghost$_{sb}$ A L R W) = False
⟨*proof*⟩

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

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

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

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

lemma is-Write$_{sb}$-simps [simp]:
is-Write$_{sb}$ (Write$_{sb}$ volatile a sop v A L R W) = True
is-Write$_{sb}$ (Read$_{sb}$ volatile a t v) = False
is-Write$_{sb}$ (Prog$_{sb}$ p p′ is) = False
is-Write$_{sb}$ (Ghost$_{sb}$ A L R W) = False
⟨*proof*⟩

lemma is-Read$_{sb}$-simps [simp]:
is-Read$_{sb}$ (Read$_{sb}$ volatile a t v) = True
is-Read$_{sb}$ (Write$_{sb}$ volatile a sop v A L R W) = False
is-Read$_{sb}$ (Prog$_{sb}$ p p′ is) = False
is-Read$_{sb}$ (Ghost$_{sb}$ A L R W) = False
⟨*proof*⟩

lemma is-Prog$_{\mathsf{sb}}$-simps [simp]:
is-Prog$_{\mathsf{sb}}$ (Read$_{\mathsf{sb}}$ volatile a t v) = False
is-Prog$_{\mathsf{sb}}$ (Write$_{\mathsf{sb}}$ volatile a sop v A L R W) = False
is-Prog$_{\mathsf{sb}}$ (Prog$_{\mathsf{sb}}$ p p′ is) = True
is-Prog$_{\mathsf{sb}}$ (Ghost$_{\mathsf{sb}}$ A L R W) = False
⟨*proof*⟩

lemma is-Ghost$_{\mathsf{sb}}$-simps [simp]:
is-Ghost$_{\mathsf{sb}}$ (Read$_{\mathsf{sb}}$ volatile a t v) = False
is-Ghost$_{\mathsf{sb}}$ (Write$_{\mathsf{sb}}$ volatile a sop v A L R W) = False
is-Ghost$_{\mathsf{sb}}$ (Prog$_{\mathsf{sb}}$ p p′ is) = False
is-Ghost$_{\mathsf{sb}}$ (Ghost$_{\mathsf{sb}}$ A L R W) = True
⟨*proof*⟩

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

lemma is-volatile-Write$_{\mathsf{sb}}$-simps [simp]:
is-volatile-Write$_{\mathsf{sb}}$ (Write$_{\mathsf{sb}}$ volatile a sop v A L R W) = volatile
is-volatile-Write$_{\mathsf{sb}}$ (Read$_{\mathsf{sb}}$ volatile a t v) = False
is-volatile-Write$_{\mathsf{sb}}$ (Prog$_{\mathsf{sb}}$ p p′ is) = False
is-volatile-Write$_{\mathsf{sb}}$ (Ghost$_{\mathsf{sb}}$ A L R W) = False
⟨*proof*⟩

lemma is-volatile-Write$_{\mathsf{sb}}$-address-of [simp]: is-volatile-Write$_{\mathsf{sb}}$ x ⟹ address-of x ≠ {}
⟨*proof*⟩

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

lemma is-volatile-Read$_{\mathsf{sb}}$-simps [simp]:
is-volatile-Read$_{\mathsf{sb}}$ (Read$_{\mathsf{sb}}$ volatile a t v) = volatile
is-volatile-Read$_{\mathsf{sb}}$ (Write$_{\mathsf{sb}}$ volatile a sop v A L R W) = False
is-volatile-Read$_{\mathsf{sb}}$ (Prog$_{\mathsf{sb}}$ p p′ is) = False
is-volatile-Read$_{\mathsf{sb}}$ (Ghost$_{\mathsf{sb}}$ A L R W) = False
⟨*proof*⟩

definition is-non-volatile-Write$_{\mathsf{sb}}$:: ′p memref ⇒ bool
where
is-non-volatile-Write$_{\mathsf{sb}}$ r = (case r of Write$_{\mathsf{sb}}$ volatile a - v - - - -⇒ ¬ volatile | - ⇒ False)

lemma is-non-volatile-Write$_{\mathsf{sb}}$-simps [simp]:
is-non-volatile-Write$_{\mathsf{sb}}$ (Write$_{\mathsf{sb}}$ volatile a sop v A L R W) = (¬ volatile)
is-non-volatile-Write$_{\mathsf{sb}}$ (Read$_{\mathsf{sb}}$ volatile a t v) = False
is-non-volatile-Write$_{\mathsf{sb}}$ (Prog$_{\mathsf{sb}}$ p p′ is) = False
is-non-volatile-Write$_{\mathsf{sb}}$ (Ghost$_{\mathsf{sb}}$ A L R W) = False
⟨*proof*⟩

definition is-non-volatile-$\text{Read}_{\mathsf{sb}}$:: $'$p memref $\Rightarrow$ bool
where
is-non-volatile-$\text{Read}_{\mathsf{sb}}$ r = (case r of $\text{Read}_{\mathsf{sb}}$ volatile a t v $\Rightarrow$ $\neg$ volatile | - $\Rightarrow$ False)

lemma is-non-volatile-$\text{Read}_{\mathsf{sb}}$-simps [simp]:
is-non-volatile-$\text{Read}_{\mathsf{sb}}$ ($\text{Read}_{\mathsf{sb}}$ volatile a t v) = ($\neg$ volatile)
is-non-volatile-$\text{Read}_{\mathsf{sb}}$ ($\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W) = False
is-non-volatile-$\text{Read}_{\mathsf{sb}}$ ($\text{Prog}_{\mathsf{sb}}$ p p$'$ is) = False
is-non-volatile-$\text{Read}_{\mathsf{sb}}$ ($\text{Ghost}_{\mathsf{sb}}$ A L R W) = False
⟨*proof*⟩

lemma is-volatile-split: is-volatile r =
(is-volatile-$\text{Read}_{\mathsf{sb}}$ r $\vee$ is-volatile-$\text{Write}_{\mathsf{sb}}$ r)
⟨*proof*⟩

lemma is-non-volatile-split:
$\neg$ is-volatile r = (is-non-volatile-$\text{Read}_{\mathsf{sb}}$ r $\vee$ is-non-volatile-$\text{Write}_{\mathsf{sb}}$ r $\vee$ is-$\text{Prog}_{\mathsf{sb}}$ r $\vee$ is-$\text{Ghost}_{\mathsf{sb}}$ r)
⟨*proof*⟩

fun outstanding-refs:: ($'$p memref $\Rightarrow$ bool) $\Rightarrow$ $'$p memref list $\Rightarrow$ addr set
where
outstanding-refs P [] = {}
| outstanding-refs P (r#rs) = (if P r then (address-of r) $\cup$ (outstanding-refs P rs)
else outstanding-refs P rs)

lemma outstanding-refs-conv: outstanding-refs P sb = $\bigcup$(address-of ` {r. r $\in$ set sb $\wedge$ P r})
⟨*proof*⟩

lemma outstanding-refs-append:
$\bigwedge$ys. outstanding-refs vol (xs@ys) = outstanding-refs vol xs $\cup$ outstanding-refs vol ys
⟨*proof*⟩

lemma outstanding-refs-empty-negate: (outstanding-refs P sb = {}) $\Longrightarrow$
(outstanding-refs (Not $\circ$ P) sb = $\bigcup$(address-of ` set sb))
⟨*proof*⟩

lemma outstanding-refs-mono-pred:
$\bigwedge$sb sb$'$.
$\forall$r. P r $\longrightarrow$ P$'$ r $\Longrightarrow$ outstanding-refs P sb $\subseteq$ outstanding-refs P$'$ sb
⟨*proof*⟩

lemma outstanding-refs-mono-set:
$\bigwedge$sb sb$'$.
set sb $\subseteq$ set sb$'$ $\Longrightarrow$ outstanding-refs P sb $\subseteq$ outstanding-refs P sb$'$
⟨*proof*⟩

lemma outstanding-refs-takeWhile:
outstanding-refs P (takeWhile P′ sb) ⊆ outstanding-refs P sb
⟨*proof*⟩

lemma outstanding-refs-subsets:
outstanding-refs is-volatile-Write$_{sb}$ sb ⊆ outstanding-refs is-Write$_{sb}$ sb
outstanding-refs is-non-volatile-Write$_{sb}$ sb ⊆ outstanding-refs is-Write$_{sb}$ sb

outstanding-refs is-volatile-Read$_{sb}$ sb ⊆ outstanding-refs is-Read$_{sb}$ sb
outstanding-refs is-non-volatile-Read$_{sb}$ sb ⊆ outstanding-refs is-Read$_{sb}$ sb

outstanding-refs is-non-volatile-Write$_{sb}$ sb ⊆ outstanding-refs (Not ∘ is-volatile) sb
outstanding-refs is-non-volatile-Read$_{sb}$ sb ⊆ outstanding-refs (Not ∘ is-volatile) sb

outstanding-refs is-volatile-Write$_{sb}$ sb ⊆ outstanding-refs (is-volatile) sb
outstanding-refs is-volatile-Read$_{sb}$ sb ⊆ outstanding-refs (is-volatile) sb

outstanding-refs is-non-volatile-Write$_{sb}$ sb ⊆ outstanding-refs (Not ∘ is-volatile-Write$_{sb}$) sb
outstanding-refs is-non-volatile-Read$_{sb}$ sb ⊆ outstanding-refs (Not ∘ is-volatile-Write$_{sb}$) sb
outstanding-refs is-volatile-Read$_{sb}$ sb ⊆ outstanding-refs (Not ∘ is-volatile-Write$_{sb}$) sb
outstanding-refs is-Read$_{sb}$ sb ⊆ outstanding-refs (Not ∘ is-volatile-Write$_{sb}$) sb
⟨*proof*⟩

lemma outstanding-non-volatile-refs-conv:
outstanding-refs (Not ∘ is-volatile) sb =
outstanding-refs is-non-volatile-Write$_{sb}$ sb ∪ outstanding-refs is-non-volatile-Read$_{sb}$ sb
⟨*proof*⟩

lemma outstanding-volatile-refs-conv:
outstanding-refs is-volatile sb =
outstanding-refs is-volatile-Write$_{sb}$ sb ∪ outstanding-refs is-volatile-Read$_{sb}$ sb
⟨*proof*⟩

lemma outstanding-is-Write$_{sb}$-refs-conv:
outstanding-refs is-Write$_{sb}$ sb =
outstanding-refs is-non-volatile-Write$_{sb}$ sb ∪ outstanding-refs is-volatile-Write$_{sb}$ sb
⟨*proof*⟩

lemma outstanding-is-Read$_{sb}$-refs-conv:
outstanding-refs is-Read$_{sb}$ sb =
outstanding-refs is-non-volatile-Read$_{sb}$ sb ∪ outstanding-refs is-volatile-Read$_{sb}$ sb
⟨*proof*⟩

lemma outstanding-not-volatile-Read$_{sb}$-refs-conv: outstanding-refs (Not ∘ is-volatile-Read$_{sb}$) sb =

outstanding-refs is-Write$_{sb}$ sb ∪ outstanding-refs is-non-volatile-Read$_{sb}$ sb
⟨*proof*⟩

lemmas misc-outstanding-refs-convs = outstanding-non-volatile-refs-conv outstanding-volatile-refs-conv
outstanding-is-Write$_{sb}$-refs-conv outstanding-is-Read$_{sb}$-refs-conv outstanding-not-volatile-Read$_{sb}$-refs-conv

lemma no-outstanding-vol-write-takeWhile-append: outstanding-refs is-volatile-Write$_{sb}$ sb = {} $\Longrightarrow$
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 ≠ {} $\Longrightarrow$
takeWhile (Not ∘ is-volatile-Write$_{sb}$) (sb@xs) = (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb)
⟨*proof*⟩

lemma no-outstanding-vol-write-dropWhile-append: outstanding-refs is-volatile-Write$_{sb}$ sb = {} $\Longrightarrow$
dropWhile (Not ∘ is-volatile-Write$_{sb}$) (sb@xs) = (dropWhile (Not ∘ is-volatile-Write$_{sb}$) xs)
⟨*proof*⟩

lemma outstanding-vol-write-dropWhile-append: outstanding-refs is-volatile-Write$_{sb}$ sb ≠ {} $\Longrightarrow$
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 = {} $\Longrightarrow$ (dropWhile (Not ∘ is-volatile-Write$_{sb}$) sb) = []
⟨*proof*⟩

lemma takeWhile-not-vol-write-outstanding-refs:
  outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) = {}
⟨*proof*⟩

lemma no-volatile-Write$_{\mathsf{sb}}$s-conv: (outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb = {}) =
  ($\forall$ r $\in$ set sb. ($\forall$ v′ sop′ a′ A L R W. r $\neq$ Write$_{\mathsf{sb}}$ True a′ sop′ v′ A L R W))
⟨*proof*⟩

lemma no-volatile-Read$_{\mathsf{sb}}$s-conv: (outstanding-refs is-volatile-Read$_{\mathsf{sb}}$ sb = {}) =
  ($\forall$ r $\in$ set sb. ($\forall$ v′ t′ a′. r $\neq$ Read$_{\mathsf{sb}}$ True a′ t′ v′))
⟨*proof*⟩

inductive sb-memop-step :: (instrs × tmps × ′p store-buffer × memory × ′dirty × ′owns × ′rels × ′shared) ⇒
  (instrs × tmps × ′p store-buffer × memory × ′dirty × ′owns × ′rels × ′shared) ⇒ bool
  (‹- $\rightarrow_{\mathsf{sb}}$ -› [60,60] 100)
where
  SBReadBuffered:
  ⟦buffered-val sb a = Some v⟧
  $\Longrightarrow$
  (Read volatile a t # is,j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$
    (is, j (t↦v), sb, m,$\mathcal{D}$, $\mathcal{O}$,$\mathcal{R}$, $\mathcal{S}$)

| SBReadUnbuffered:
  ⟦buffered-val sb a = None⟧
  $\Longrightarrow$
  (Read volatile a t # is, j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$
    (is, j (t↦m a), sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBWriteNonVolatile:
  (Write False a (D,f) A L R W#is, j, sb, m,$\mathcal{D}$,$\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$
    (is, j, sb@ [Write$_{\mathsf{sb}}$ False a (D,f) (f j) A L R W], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBWriteVolatile:

  (Write True a (D,f) A L R W# is, j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$
    (is, j, sb@[Write$_{\mathsf{sb}}$ True a (D,f) (f j) A L R W], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBFence:
  (Fence # is, j, [], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$ (is, j, [], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBRMWReadOnly:
  ⟦¬ cond (j(t↦m a))⟧ $\Longrightarrow$
  (RMW a t (D,f) cond ret A L R W# is, j, [], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$ (is, j(t↦m a),[], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBRMWWrite:
⟦cond (j(t↦m a))⟧ ⟹
(RMW a t (D,f) cond ret A L R W# is, j, [], m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$
(is, j(t↦ret (m a) (f(j(t↦m a)))),[], m(a:= f(j(t↦m a))),$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBGhost:
(Ghost A L R W# is, j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sb}}$
(is, j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

inductive sbh-memop-step ::
(instrs × tmps × ′p store-buffer × memory × bool × owns × rels × shared
) ⇒
(instrs × tmps × ′p store-buffer × memory × bool × owns × rels × shared
) ⇒ bool
(‹- $\rightarrow_{\mathsf{sbh}}$ -› [60,60] 100)
where
SBHReadBuffered:
⟦buffered-val sb a = Some v⟧
⟹
(Read volatile a t # is, j, sb, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$
(is, j (t↦v), sb@[$\text{Read}_{\mathsf{sb}}$ volatile a t v], m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

| SBHReadUnbuffered:
⟦buffered-val sb a = None⟧
⟹
(Read volatile a t # is, j, sb, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$
(is, j (t↦m a), sb@[$\text{Read}_{\mathsf{sb}}$ volatile a t (m a)], m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

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

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

| SBHFence:
(Fence # is, j, [], m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is, j, [], m, False, $\mathcal{O}$, Map.empty, $\mathcal{S}$)

| SBHRMWReadOnly:
⟦¬ cond (j(t↦m a))⟧ ⟹
(RMW a t (D,f) cond ret A L R W# is, j, [], m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is, j(t↦m a),[], m, False, $\mathcal{O}$, Map.empty, $\mathcal{S}$)

| SBHRMWWrite:
⟦cond (j(t↦m a))⟧ ⟹
(RMW a t (D,f) cond ret A L R W# is, j, [], m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$
(is, j(t↦ret (m a) (f(j(t↦m a)))),[], m(a:= f(j(t↦m a))), False, $\mathcal{O}$ ∪ A − R,Map.empty, $\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)

| SBHGhost:
 (Ghost A L R W# is, j, sb, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$
  (is, j, sb@[$\text{Ghost}_{\mathsf{sb}}$ A L R W], m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$, $\mathcal{S}$)

interpretation direct: memory-system direct-memop-step id-storebuffer-step ⟨*proof*⟩
interpretation sb: memory-system sb-memop-step store-buffer-step ⟨*proof*⟩
interpretation sbh: memory-system sbh-memop-step flush-step ⟨*proof*⟩

primrec non-volatile-owned-or-read-only:: bool ⇒ shared ⇒ owns ⇒ ′a memref list ⇒ bool
where
non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ [] = True
| non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ (x#xs) =
 (case x of
  $\text{Read}_{\mathsf{sb}}$ volatile a t v ⇒
   (¬volatile ⟶ pending-write ⟶ (a ∈ $\mathcal{O}$ ∨ a ∈ read-only $\mathcal{S}$)) ∧
   non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ xs
 | $\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W ⇒
   (if volatile then non-volatile-owned-or-read-only True ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) ($\mathcal{O}$ ∪ A − R) xs
   else a ∈ $\mathcal{O}$ ∧ non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ xs)
 | $\text{Ghost}_{\mathsf{sb}}$ A L R W ⇒ non-volatile-owned-or-read-only pending-write ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) ($\mathcal{O}$ ∪ A − R) xs
 | - ⇒ non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ xs)

primrec acquired :: bool ⇒ ′a memref list ⇒ addr set ⇒ addr set
where
acquired pending-write [] A = (if pending-write then A else {})
| acquired pending-write (x#xs) A =
 (case x of
  $\text{Write}_{\mathsf{sb}}$ volatile - - - A′ L R W⇒
   (if volatile then acquired True xs (if pending-write then (A ∪ A′ − R) else (A′ − R))
   else acquired pending-write xs A)
 | $\text{Ghost}_{\mathsf{sb}}$ A′ L R W ⇒ acquired pending-write xs (if pending-write then (A ∪ A′ − R) else A)
 | - ⇒ acquired pending-write xs A)

primrec share :: ′a memref list ⇒ shared ⇒ shared
where
share [] S = S
| share (x#xs) S =
 (case x of
  $\text{Write}_{\mathsf{sb}}$ volatile - - - A L R W ⇒ (if volatile then (share xs (S $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)) else share xs S)
 | $\text{Ghost}_{\mathsf{sb}}$ A L R W ⇒ share xs (S $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)
 | - ⇒ share xs S)

primrec acquired-reads :: bool ⇒ 'a memref list ⇒ addr set ⇒ addr set
where
acquired-reads pending-write [] A = {}
| acquired-reads pending-write (x#xs) A =
 (case x of
  $\text{Read}_{\mathsf{sb}}$ volatile a t v ⇒ (if pending-write ∧ ¬ volatile ∧ a ∈ A
   then insert a (acquired-reads pending-write xs A)
   else acquired-reads pending-write xs A)
 | $\text{Write}_{\mathsf{sb}}$ volatile - - - A' L R W ⇒
  (if volatile then acquired-reads True xs (if pending-write then (A ∪ A' − R) else (A' − R))
  else acquired-reads pending-write xs A)
 | $\text{Ghost}_{\mathsf{sb}}$ 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 ∨ outstanding-refs is-volatile-$\text{Write}_{\mathsf{sb}}$ xs ≠ {}) ys (acquired pending-write xs A)
⟨*proof*⟩

lemma acquired-take-drop:
acquired (pending-write ∨ outstanding-refs is-volatile-Write$_{sb}$ (takeWhile P xs) ≠ {})
(dropWhile P xs) (acquired pending-write (takeWhile P xs) A) =
acquired pending-write xs A
⟨*proof*⟩

lemma share-mono: ⋀A B. dom A ⊆ dom B ⟹ dom (share xs A) ⊆ dom (share xs B)
⟨*proof*⟩

lemma share-mono-in:
assumes x-in: x ∈ dom (share xs A)
assumes sub: dom A ⊆ dom B
shows x ∈ dom (share xs B)
⟨*proof*⟩

lemma acquired-reads-mono:
⋀A B pending-write. A ⊆ B ⟹ acquired-reads pending-write xs A ⊆ acquired-reads pending-write xs B
⟨*proof*⟩

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

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

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

lemma acquired-reads-pending-write-mono:
⋀A. acquired-reads False xs A ⊆ acquired-reads True xs A
⟨*proof*⟩

lemma acquired-reads-pending-write-mono-in:
assumes x-in: x ∈ acquired-reads False xs A
shows x ∈ acquired-reads True xs A
⟨*proof*⟩

lemma acquired-reads-append: ⋀pending-write A. acquired-reads pending-write (xs@ys) A =
acquired-reads pending-write xs A ∪
acquired-reads (pending-write ∨ (outstanding-refs is-volatile-Write$_{sb}$ xs ≠ {})) ys
(acquired pending-write xs A)
⟨*proof*⟩

lemma in-acquired-reads-no-pending-write-outstanding-write:
$\bigwedge$A. a $\in$ acquired-reads False xs A $\Longrightarrow$ outstanding-refs (is-volatile-Write$_{\mathsf{sb}}$) xs $\neq$ {}
⟨*proof*⟩

lemma augment-read-only-mono: read-only $\mathcal{S}$ $\subseteq$ read-only $\mathcal{S}'$ $\Longrightarrow$
read-only ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R) $\subseteq$ read-only ($\mathcal{S}'$ $\oplus_{\mathsf{W}}$ R)
⟨*proof*⟩

lemma restrict-read-only-mono: read-only $\mathcal{S}$ $\subseteq$ read-only $\mathcal{S}'$ $\Longrightarrow$
read-only ($\mathcal{S}$ $\ominus_{\mathsf{A}}$ L) $\subseteq$ read-only ($\mathcal{S}'$ $\ominus_{\mathsf{A}}$ L)
⟨*proof*⟩

lemma share-read-only-mono: $\bigwedge\mathcal{S}$ $\mathcal{S}'$. read-only $\mathcal{S}$ $\subseteq$ read-only $\mathcal{S}'$ $\Longrightarrow$
read-only (share sb $\mathcal{S}$) $\subseteq$ read-only (share sb $\mathcal{S}'$)
⟨*proof*⟩

lemma non-volatile-owned-or-read-only-append:
$\bigwedge\mathcal{O}$ $\mathcal{S}$ pending-write. non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ (xs@ys)
= (non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ xs $\wedge$
non-volatile-owned-or-read-only (pending-write $\vee$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ xs $\neq$ {})
(share xs $\mathcal{S}$) (acquired True xs $\mathcal{O}$) ys)
⟨*proof*⟩

lemma non-volatile-owned-or-read-only-mono:
$\bigwedge\mathcal{O}$ $\mathcal{O}'$ $\mathcal{S}$ pending-write. $\mathcal{O}$ $\subseteq$ $\mathcal{O}'$ $\Longrightarrow$ non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ xs
$\Longrightarrow$ non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}'$ xs
⟨*proof*⟩

lemma non-volatile-owned-or-read-only-shared-mono:
$\bigwedge\mathcal{S}$ $\mathcal{S}'$ $\mathcal{O}$ pending-write. $\mathcal{S}$ $\subseteq_{\mathsf{s}}$ $\mathcal{S}'$ $\Longrightarrow$ non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ xs
$\Longrightarrow$ non-volatile-owned-or-read-only pending-write $\mathcal{S}'$ $\mathcal{O}$ xs
⟨*proof*⟩

lemma non-volatile-owned-or-read-only-pending-write-antimono:
$\bigwedge\mathcal{O}$ $\mathcal{S}$. non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ xs
$\Longrightarrow$ non-volatile-owned-or-read-only False $\mathcal{S}$ $\mathcal{O}$ xs
⟨*proof*⟩

primrec all-acquired :: 'a memref list $\Rightarrow$ addr set
where
all-acquired [] = {}
| all-acquired (i#is) =
(case i of

$\text{Write}_{\mathsf{sb}}$ volatile - - - A L R W ⇒ (if volatile then A ∪ all-acquired is else all-acquired is)
| $\text{Ghost}_{\mathsf{sb}}$ A L R W ⇒ A ∪ all-acquired is
| - ⇒ all-acquired is)

lemma all-acquired-append: all-acquired (xs@ys) = all-acquired xs ∪ all-acquired ys
⟨*proof*⟩

lemma acquired-reads-all-acquired: $\bigwedge\mathcal{O}$ pending-write.
acquired-reads pending-write sb $\mathcal{O} \subseteq \mathcal{O}$ ∪ all-acquired sb
⟨*proof*⟩

lemma acquired-takeWhile-non-volatile-$\text{Write}_{\mathsf{sb}}$:
⋀A. (acquired True (takeWhile (Not ∘ is-volatile-$\text{Write}_{\mathsf{sb}}$) sb) A) ⊆
A ∪ all-acquired (takeWhile (Not ∘ is-volatile-$\text{Write}_{\mathsf{sb}}$) sb)
⟨*proof*⟩

lemma acquired-False-takeWhile-non-volatile-$\text{Write}_{\mathsf{sb}}$:
acquired False (takeWhile (Not ∘ is-volatile-$\text{Write}_{\mathsf{sb}}$) sb) A = {}
⟨*proof*⟩

lemma outstanding-refs-takeWhile-opposite: outstanding-refs P (takeWhile (Not ∘ P) xs) = {}
⟨*proof*⟩

lemma no-outstanding-volatile-$\text{Write}_{\mathsf{sb}}$-acquired:
outstanding-refs is-volatile-$\text{Write}_{\mathsf{sb}}$ sb = {} ⟹ acquired False sb A = {}
⟨*proof*⟩

lemma acquired-all-acquired:⋀pending-write A. acquired pending-write xs A ⊆ A ∪ all-acquired xs
⟨*proof*⟩

lemma acquired-all-acquired-in: x ∈ acquired pending-write xs A ⟹ x ∈ A ∪ all-acquired xs
⟨*proof*⟩

primrec sharing-consistent:: shared ⇒ owns ⇒ ′a memref list ⇒ bool
where
sharing-consistent $\mathcal{S}$ $\mathcal{O}$ [] = True
| sharing-consistent $\mathcal{S}$ $\mathcal{O}$ (r#rs) =
(case r of
$\text{Write}_{\mathsf{sb}}$ volatile - - - A L R W ⇒
(if volatile then A ⊆ dom $\mathcal{S} \cup \mathcal{O}$ ∧ L ⊆ A ∧ A ∩ R = {} ∧ R ⊆ $\mathcal{O}$ ∧
sharing-consistent ($\mathcal{S} \oplus_{\mathsf{W}} R \ominus_{\mathsf{A}} L$) ($\mathcal{O}$ ∪ A − R) rs
else sharing-consistent $\mathcal{S}$ $\mathcal{O}$ rs)

| $\mathrm{Ghost}_{\mathsf{sb}}$ A L R W $\Rightarrow$ A $\subseteq$ dom $\mathcal{S} \cup \mathcal{O} \wedge$ L $\subseteq$ A $\wedge$ A $\cap$ R = {} $\wedge$ R $\subseteq \mathcal{O} \wedge$
sharing-consistent ($\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) ($\mathcal{O} \cup$ A $-$ R) rs
| - $\Rightarrow$ sharing-consistent $\mathcal{S}$ $\mathcal{O}$ rs)

lemma sharing-consistent-all-acquired:
$\bigwedge\mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ all-acquired sb $\subseteq$ dom $\mathcal{S} \cup \mathcal{O}$
⟨*proof*⟩

lemma sharing-consistent-append:
$\bigwedge\mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ (xs@ys) =
(sharing-consistent $\mathcal{S}$ $\mathcal{O}$ xs $\wedge$
sharing-consistent (share xs $\mathcal{S}$) (acquired True xs $\mathcal{O}$) ys)
⟨*proof*⟩

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

lemma read-only-reads-append:
$\bigwedge\mathcal{O}$. read-only-reads $\mathcal{O}$ (xs@ys) =
read-only-reads $\mathcal{O}$ xs $\cup$ read-only-reads (acquired True xs $\mathcal{O}$) ys
⟨*proof*⟩

lemma read-only-reads-antimono:
$\bigwedge\mathcal{O}\ \mathcal{O}'$.
$\mathcal{O} \subseteq \mathcal{O}' \Longrightarrow$ read-only-reads $\mathcal{O}'$ sb $\subseteq$ read-only-reads $\mathcal{O}$ sb
⟨*proof*⟩

primrec non-volatile-writes-unshared:: shared $\Rightarrow$ 'a memref list $\Rightarrow$ bool
where
non-volatile-writes-unshared $\mathcal{S}$ [] = True
| non-volatile-writes-unshared $\mathcal{S}$ (x#xs) =
(case x of
$\mathrm{Write}_{\mathsf{sb}}$ volatile a sop v A L R W $\Rightarrow$ (if volatile then non-volatile-writes-unshared ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) xs
else a $\notin$ dom $\mathcal{S}$ $\wedge$ non-volatile-writes-unshared $\mathcal{S}$ xs)
| $\mathrm{Ghost}_{\mathsf{sb}}$ A L R W $\Rightarrow$ non-volatile-writes-unshared ($\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) xs
| - $\Rightarrow$ non-volatile-writes-unshared $\mathcal{S}$ xs)

lemma non-volatile-writes-unshared-append:
$\bigwedge\mathcal{S}$. non-volatile-writes-unshared $\mathcal{S}$ (xs@ys)
= (non-volatile-writes-unshared $\mathcal{S}$ xs $\wedge$ non-volatile-writes-unshared (share xs $\mathcal{S}$) ys)
$\langle proof \rangle$

lemma non-volatile-writes-unshared-antimono:
$\bigwedge\mathcal{S}\ \mathcal{S}'$. dom $\mathcal{S} \subseteq$ dom $\mathcal{S}' \Longrightarrow$ non-volatile-writes-unshared $\mathcal{S}'$ xs
$\Longrightarrow$ non-volatile-writes-unshared $\mathcal{S}$ xs
$\langle proof \rangle$

primrec no-write-to-read-only-memory:: shared $\Rightarrow$ 'a memref list $\Rightarrow$ bool
where
no-write-to-read-only-memory $\mathcal{S}$ [] = True
| no-write-to-read-only-memory $\mathcal{S}$ (x#xs) =
(case x of
$\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W $\Rightarrow$ a $\notin$ read-only $\mathcal{S}$ $\wedge$
(if volatile then no-write-to-read-only-memory ($\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) xs
else no-write-to-read-only-memory $\mathcal{S}$ xs)
| $\text{Ghost}_{\mathsf{sb}}$ A L R W $\Rightarrow$ no-write-to-read-only-memory ($\mathcal{S} \oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) xs
| - $\Rightarrow$ no-write-to-read-only-memory $\mathcal{S}$ xs)

lemma no-write-to-read-only-memory-append:
$\bigwedge\mathcal{S}$. no-write-to-read-only-memory $\mathcal{S}$ (xs@ys)
= (no-write-to-read-only-memory $\mathcal{S}$ xs $\wedge$ no-write-to-read-only-memory (share xs $\mathcal{S}$) ys)
$\langle proof \rangle$

lemma no-write-to-read-only-memory-antimono:
$\bigwedge\mathcal{S}\ \mathcal{S}'$. $\mathcal{S} \subseteq_{\mathsf{s}} \mathcal{S}' \Longrightarrow$ no-write-to-read-only-memory $\mathcal{S}'$ xs
$\Longrightarrow$ no-write-to-read-only-memory $\mathcal{S}$ xs
$\langle proof \rangle$

locale outstanding-non-volatile-refs-owned-or-read-only =
fixes $\mathcal{S}$::shared
fixes ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes outstanding-non-volatile-refs-owned-or-read-only:
$\bigwedge$i is $\mathcal{O}\ \mathcal{R}\ \mathcal{D}$ j sb p.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$
$\Longrightarrow$
non-volatile-owned-or-read-only False $\mathcal{S}\ \mathcal{O}$ 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:
$\bigwedge$i p_i is_i $\mathcal{O}_i$ $\mathcal{R}_i$ $\mathcal{D}_i$ j_i sb_i j p_j is_j $\mathcal{O}_j$ $\mathcal{R}_j$ $\mathcal{D}_j$ j_j sb_j.
$[\![$i < length ts; j < length ts; i$\neq$j;
ts!i = (p_i,is_i,j_i,sb_i,$\mathcal{D}_i$,$\mathcal{O}_i$,$\mathcal{R}_i$); ts!j = (p_j,is_j,j_j,sb_j,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$)
$]\!]$

$\Longrightarrow$
$(\mathcal{O}_j \cup \text{all-acquired } sb_j) \cap \text{outstanding-refs is-volatile-Write}_{sb}\ sb_i = \{\}$

locale read-only-reads-unowned =
fixes ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes read-only-reads-unowned:
$\bigwedge i\ p_i\ is_i\ \mathcal{O}_i\ \mathcal{R}_i\ \mathcal{D}_i\ j_i\ sb_i\ j\ p_j\ is_j\ \mathcal{O}_j\ \mathcal{R}_j\ \mathcal{D}_j\ j_j\ sb_j.$
$[\![i < \text{length ts};\ j < \text{length ts};\ i \neq j;$
$ts!i = (p_i,is_i,j_i,sb_i,\mathcal{D}_i,\mathcal{O}_i,\mathcal{R}_i);\ ts!j = (p_j,is_j,j_j,sb_j,\mathcal{D}_j,\mathcal{O}_j,\mathcal{R}_j)$
$]\!]$
$\Longrightarrow$
$(\mathcal{O}_j \cup \text{all-acquired } sb_j) \cap$
read-only-reads (acquired True
(takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb_i) $\mathcal{O}_i$)
(dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb_i) = {}

locale ownership-distinct =
fixes ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes ownership-distinct:
$\bigwedge i\ j\ p_i\ is_i\ \mathcal{O}_i\ \mathcal{R}_i\ \mathcal{D}_i\ j_i\ sb_i\ p_j\ is_j\ \mathcal{O}_j\ \mathcal{R}_j\ \mathcal{D}_j\ j_j\ sb_j.$
$[\![i < \text{length ts};\ j < \text{length ts};\ i \neq j;$
$ts!i = (p_i,is_i,j_i,sb_i,\mathcal{D}_i,\mathcal{O}_i,\mathcal{R}_i);\ ts!j = (p_j,is_j,j_j,sb_j,\mathcal{D}_j,\mathcal{O}_j,\mathcal{R}_j)$
$]\!] \Longrightarrow (\mathcal{O}_i \cup \text{all-acquired } sb_i) \cap (\mathcal{O}_j \cup \text{all-acquired } sb_j) = \{\}$

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

locale outstanding-non-volatile-writes-unshared =
fixes $\mathcal{S}$::shared **and** ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes outstanding-non-volatile-writes-unshared:
$\bigwedge i\ p\ is\ \mathcal{O}\ \mathcal{R}\ \mathcal{D}\ j\ sb.$
$[\![i < \text{length ts};\ ts!i = (p,is,j,sb,\mathcal{D},\mathcal{O},\mathcal{R})\ ]\!]$
$\Longrightarrow$
non-volatile-writes-unshared $\mathcal{S}$ sb

locale sharing-consis =
fixes $\mathcal{S}$::shared **and** ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes sharing-consis:
$\bigwedge i\ p\ is\ \mathcal{O}\ \mathcal{R}\ \mathcal{D}\ j\ sb.$
$[\![i < \text{length ts};\ ts!i = (p,is,j,sb,\mathcal{D},\mathcal{O},\mathcal{R})\ ]\!]$
$\Longrightarrow$
sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb

locale no-outstanding-write-to-read-only-memory =
fixes $\mathcal{S}$::shared **and** ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes no-outstanding-write-to-read-only-memory:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$
$\Longrightarrow$
no-write-to-read-only-memory $\mathcal{S}$ 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:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
read-only-reads (acquired True (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb') $\mathcal{O}'$)
(dropWhile (Not ∘ is-volatile-Write$_{sb}$) sb') ⊆ read-only-reads (acquired True (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) $\mathcal{O}$)
(dropWhile (Not ∘ is-volatile-Write$_{sb}$) sb);
$\mathcal{O}'$ ∪ all-acquired sb' ⊆ $\mathcal{O}$ ∪ all-acquired sb$]\!]$ $\Longrightarrow$
read-only-reads-unowned (ts[i := (p',is',j',sb',$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma outstanding-non-volatile-refs-owned-or-read-only-tl:
outstanding-non-volatile-refs-owned-or-read-only $\mathcal{S}$ (t#ts) $\Longrightarrow$ outstanding-non-volatile-refs-owned-or-read-only $\mathcal{S}$ ts
⟨*proof*⟩

lemma outstanding-volatile-writes-unowned-by-others-tl:
outstanding-volatile-writes-unowned-by-others (t#ts) $\Longrightarrow$ outstanding-volatile-writes-unowned-by-others ts
⟨*proof*⟩

lemma read-only-reads-unowned-tl:
read-only-reads-unowned (t # ts) $\Longrightarrow$

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 $\mathcal{S}$ (t#ts) $\Longrightarrow$ valid-ownership $\mathcal{S}$ ts
⟨proof⟩

lemma sharing-consistent-takeWhile:
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows sharing-consistent $\mathcal{S}$ $\mathcal{O}$ (takeWhile P sb)
⟨proof⟩

lemma sharing-consis-tl: sharing-consis $\mathcal{S}$ (t#ts) $\Longrightarrow$ sharing-consis $\mathcal{S}$ ts
⟨proof⟩

lemma sharing-consis-Cons:
⟦sharing-consis $\mathcal{S}$ ts; sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb⟧
$\Longrightarrow$ sharing-consis $\mathcal{S}$ ((p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)#ts)
⟨proof⟩

lemma outstanding-non-volatile-writes-unshared-tl:
outstanding-non-volatile-writes-unshared $\mathcal{S}$ (t#ts) $\Longrightarrow$
outstanding-non-volatile-writes-unshared $\mathcal{S}$ ts
⟨proof⟩

lemma no-outstanding-write-to-read-only-memory-tl:
no-outstanding-write-to-read-only-memory $\mathcal{S}$ (t#ts) $\Longrightarrow$
no-outstanding-write-to-read-only-memory $\mathcal{S}$ ts
⟨proof⟩

lemma valid-ownership-and-sharing-tl:
valid-ownership-and-sharing $\mathcal{S}$ (t#ts) $\Longrightarrow$ valid-ownership-and-sharing $\mathcal{S}$ ts
⟨proof⟩

lemma non-volatile-owned-or-read-only-outstanding-non-volatile-writes:
$\bigwedge\mathcal{O}$ $\mathcal{S}$ pending-write. ⟦non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb⟧
$\Longrightarrow$
outstanding-refs is-non-volatile-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
⟨proof⟩

lemma (**in** outstanding-non-volatile-refs-owned-or-read-only) outstanding-non-volatile-writes-owned:

assumes i-bound: $i < \text{length } ts$
assumes ts-i: $ts!i = (p,is,j,sb,\mathcal{D},\mathcal{O},\mathcal{R})$
shows outstanding-refs is-non-volatile-Write$_{\mathsf{sb}}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb
⟨*proof*⟩

lemma non-volatile-reads-acquired-or-read-only:
$\bigwedge\mathcal{O}\ \mathcal{S}$. ⟦non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb; sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb⟧
$\Longrightarrow$
outstanding-refs is-non-volatile-Read$_{\mathsf{sb}}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb $\cup$ read-only $\mathcal{S}$
⟨*proof*⟩

lemma non-volatile-reads-acquired-or-read-only-reads:
$\bigwedge\mathcal{O}\ \mathcal{S}$ pending-write. ⟦non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb⟧
$\Longrightarrow$
outstanding-refs is-non-volatile-Read$_{\mathsf{sb}}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb $\cup$ read-only-reads $\mathcal{O}$ sb
⟨*proof*⟩

lemma non-volatile-owned-or-read-only-outstanding-refs:
$\bigwedge\mathcal{O}\ \mathcal{S}$ pending-write. ⟦non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb⟧
$\Longrightarrow$
outstanding-refs (Not $\circ$ is-volatile) sb $\subseteq \mathcal{O} \cup$ all-acquired sb $\cup$ read-only-reads $\mathcal{O}$ sb
⟨*proof*⟩

lemma no-unacquired-write-to-read-only:
$\bigwedge\mathcal{S}\ \mathcal{O}$. ⟦no-write-to-read-only-memory $\mathcal{S}$ sb;sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb;
a $\in$ read-only $\mathcal{S}$; a $\notin$ ($\mathcal{O} \cup$ all-acquired sb)⟧
$\Longrightarrow$ a $\notin$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb
⟨*proof*⟩

lemma read-only-reads-read-only:
$\bigwedge\mathcal{S}\ \mathcal{O}$. ⟦non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb;
sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb⟧
$\Longrightarrow$
read-only-reads $\mathcal{O}$ sb $\subseteq \mathcal{O} \cup$ all-acquired sb $\cup$ read-only $\mathcal{S}$
⟨*proof*⟩

lemma no-unacquired-write-to-read-only-reads:
$\bigwedge\mathcal{S}\ \mathcal{O}$. ⟦no-write-to-read-only-memory $\mathcal{S}$ sb;

non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb; sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb;
a $\in$ read-only-reads $\mathcal{O}$ sb; a $\notin$ ($\mathcal{O}$ $\cup$ all-acquired sb)$\rrbracket$
$\Longrightarrow$ a $\notin$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb
$\langle proof \rangle$

lemma no-unacquired-write-to-read-only$''$:
assumes no-wrt: no-write-to-read-only-memory $\mathcal{S}$ sb
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows read-only $\mathcal{S}$ $\cap$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
$\langle proof \rangle$

lemma no-unacquired-volatile-write-to-read-only:
assumes no-wrt: no-write-to-read-only-memory $\mathcal{S}$ sb
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows read-only $\mathcal{S}$ $\cap$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
$\langle proof \rangle$

lemma no-unacquired-non-volatile-write-to-read-only-reads:
assumes no-wrt: no-write-to-read-only-memory $\mathcal{S}$ sb
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows read-only $\mathcal{S}$ $\cap$ outstanding-refs is-non-volatile-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
$\langle proof \rangle$

lemma no-unacquired-write-to-read-only-reads$'$:
assumes no-wrt: no-write-to-read-only-memory $\mathcal{S}$ sb
assumes non-vol: non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows read-only-reads $\mathcal{O}$ sb $\cap$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
$\langle proof \rangle$

lemma no-unacquired-volatile-write-to-read-only-reads:
assumes no-wrt: no-write-to-read-only-memory $\mathcal{S}$ sb
assumes non-vol: non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows read-only-reads $\mathcal{O}$ sb $\cap$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
$\langle proof \rangle$

lemma no-unacquired-non-volatile-write-to-read-only:
assumes no-wrt: no-write-to-read-only-memory $\mathcal{S}$ sb
assumes non-vol: non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows read-only-reads $\mathcal{O}$ sb $\cap$ outstanding-refs is-non-volatile-Write$_{\mathsf{sb}}$ sb $\subseteq$ $\mathcal{O}$ $\cup$ all-acquired sb
$\langle proof \rangle$

lemma set-dropWhileD: $x \in$ set (dropWhile P xs) $\Longrightarrow x \in$ set xs
⟨*proof*⟩

lemma outstanding-refs-takeWhileD:
$x \in$ outstanding-refs P (takeWhile P′ sb) $\Longrightarrow x \in$ outstanding-refs P sb
⟨*proof*⟩

lemma outstanding-refs-dropWhileD:
$x \in$ outstanding-refs P (dropWhile P′ sb) $\Longrightarrow x \in$ outstanding-refs P sb
⟨*proof*⟩

lemma dropWhile-ConsD: dropWhile P xs = y#ys $\Longrightarrow \neg$ P y
⟨*proof*⟩

lemma non-volatile-owned-or-read-only-drop:
non-volatile-owned-or-read-only False $\mathcal{S}$ $\mathcal{O}$ sb
$\Longrightarrow$ non-volatile-owned-or-read-only True
(share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{S}$)
(acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$)
(dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
⟨*proof*⟩

lemma read-only-share: $\bigwedge \mathcal{S}\ \mathcal{O}$.
sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$
read-only (share sb $\mathcal{S}$) $\subseteq$ read-only $\mathcal{S} \cup \mathcal{O} \cup$ all-acquired sb
⟨*proof*⟩

lemma (**in** valid-ownership-and-sharing) outstanding-non-write-non-vol-reads-drop-disj:
assumes i-bound: i < length ts
assumes j-bound: j < length ts
assumes neq-i-j: $i \neq j$
assumes ith: ts!i = (p_i,is_i,j_i,sb_i,$\mathcal{D}_i$,$\mathcal{O}_i$,$\mathcal{R}_i$)
assumes jth: ts!j = (p_j,is_j,j_j,sb_j,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$)
shows outstanding-refs is-Write$_{\mathsf{sb}}$ (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb_i) $\cap$
outstanding-refs is-non-volatile-Read$_{\mathsf{sb}}$ (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb_j)
= {}
⟨*proof*⟩

lemma (**in** valid-ownership-and-sharing) outstanding-non-volatile-write-disj:
assumes i-bound: i < length ts
assumes j-bound: j < length ts
assumes neq-i-j: $i \neq j$
assumes ith: ts!i = (p_i,is_i,j_i,sb_i,$\mathcal{D}_i$,$\mathcal{O}_i$,$\mathcal{R}_i$)
assumes jth: ts!j = (p_j,is_j,j_j,sb_j,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$)

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 $\mathcal{O}_j$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb$_j$)) ∪
($\mathcal{O}_j$ ∪ 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$_i$,j$_i$,sb$_i$,$\mathcal{D}_i$,$\mathcal{O}_i$,$\mathcal{R}_i$)
assumes jth: ts!j = (p$_j$,is$_j$,j$_j$,sb$_j$,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$)
shows outstanding-refs (is-non-volatile-Write$_{sb}$) (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb$_i$) ∩
outstanding-refs (Not ∘ is-volatile-Read$_{sb}$) (dropWhile (Not ∘ is-volatile-Write$_{sb}$) sb$_j$) = {}
(**is** ?non-vol-writes-i ∩ ?not-volatile-j = {})
⟨*proof*⟩

lemma (**in** valid-ownership-and-sharing) outstanding-refs-is-Write$_{sb}$-takeWhile-disj:
∀ i < length ts. (∀ j < length ts. i ≠ j ⟶
(let (-,-,-,sb$_i$,-,-,-) = ts!i;
(-,-,-,sb$_j$,-,-,-) = ts!j
in outstanding-refs is-Write$_{sb}$ sb$_i$ ∩
outstanding-refs is-Write$_{sb}$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb$_j$) = {}))
⟨*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)

$\langle proof \rangle$

lemma read-tmps-mono: $\bigwedge$ys. set xs $\subseteq$ set ys $\Longrightarrow$ read-tmps xs $\subseteq$ read-tmps ys
$\langle proof \rangle$

fun distinct-read-tmps:: 'p store-buffer $\Rightarrow$ bool
where
distinct-read-tmps [] = True
| distinct-read-tmps (r#rs) =
(case r of
Read_{sb} volatile a t v $\Rightarrow$ t $\notin$ (read-tmps rs) $\wedge$ distinct-read-tmps rs
| - $\Rightarrow$ distinct-read-tmps rs)

lemma distinct-read-tmps-conv:
distinct-read-tmps xs = ($\forall$ i < length xs. $\forall$ j < length xs. i $\neq$ j $\longrightarrow$
(case xs!i of
Read_{sb} - - t_i - $\Rightarrow$ case xs!j of Read_{sb} - - t_j - $\Rightarrow$ $t_i \neq t_j$ | - $\Rightarrow$ True
| - $\Rightarrow$ True))
— Nice lemma, ugly proof.
$\langle proof \rangle$

fun load-tmps:: instrs $\Rightarrow$ tmp set
where
load-tmps [] = {}
| load-tmps (i#is) =
(case i of
Read volatile a t $\Rightarrow$ insert t (load-tmps is)
| RMW - t - - - - - - - $\Rightarrow$ insert t (load-tmps is)
| - $\Rightarrow$ load-tmps is)

lemma in-load-tmps-conv:
(t $\in$ load-tmps xs) = (($\exists$ volatile a. Read volatile a t $\in$ set xs) $\vee$
($\exists$ a sop cond ret A L R W. RMW a t sop cond ret A L R W $\in$ set xs))
$\langle proof \rangle$

lemma load-tmps-mono: $\bigwedge$ys. set xs $\subseteq$ set ys $\Longrightarrow$ load-tmps xs $\subseteq$ load-tmps ys
$\langle proof \rangle$

fun distinct-load-tmps:: instrs $\Rightarrow$ bool
where
distinct-load-tmps [] = True
| distinct-load-tmps (r#rs) =
(case r of
Read volatile a t $\Rightarrow$ t $\notin$ (load-tmps rs) $\wedge$ distinct-load-tmps rs
| RMW a t sop cond ret A L R W $\Rightarrow$ t $\notin$ (load-tmps rs) $\wedge$ distinct-load-tmps rs
| - $\Rightarrow$ distinct-load-tmps rs)

locale load-tmps-distinct =
fixes ts::($'$p,$'$p store-buffer,bool,owns,rels) thread-config list
assumes load-tmps-distinct:
  $\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
  $[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$
  $\Longrightarrow$
  distinct-load-tmps is

locale read-tmps-distinct =
fixes ts::($'$p,$'$p store-buffer,bool,owns,rels) thread-config list
assumes read-tmps-distinct:
  $\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
  $[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$
  $\Longrightarrow$
  distinct-read-tmps sb

locale load-tmps-read-tmps-distinct =
fixes ts::($'$p,$'$p store-buffer,bool,owns,rels) thread-config list
assumes load-tmps-read-tmps-distinct:
  $\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
  $[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$
  $\Longrightarrow$
  load-tmps is ∩ read-tmps sb = {}

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

lemma rev-read-tmps: read-tmps (rev xs) = read-tmps xs
  ⟨*proof*⟩

lemma rev-load-tmps: load-tmps (rev xs) = load-tmps xs
  ⟨*proof*⟩

lemma distinct-read-tmps-append: $\bigwedge$ys. distinct-read-tmps (xs @ ys) =
    (distinct-read-tmps xs ∧ distinct-read-tmps ys ∧
    read-tmps xs ∩ read-tmps ys = {})
⟨*proof*⟩

lemma distinct-load-tmps-append: $\bigwedge$ys. distinct-load-tmps (xs @ ys) =
    (distinct-load-tmps xs ∧ distinct-load-tmps ys ∧
    load-tmps xs ∩ load-tmps ys = {})
⟨*proof*⟩

lemma read-tmps-append: read-tmps (xs@ys) = (read-tmps xs ∪ read-tmps ys)
  ⟨*proof*⟩

lemma load-tmps-append: load-tmps (xs@ys) = (load-tmps xs ∪ load-tmps ys)

⟨*proof*⟩

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

lemma in-write-sops-conv:
  (sop ∈ write-sops xs) = (∃ volatile a v A L R W. Write_{sb} volatile a sop v A L R W ∈ set xs)
  ⟨*proof*⟩

lemma write-sops-mono: ⋀ys. set xs ⊆ set ys ⟹ write-sops xs ⊆ write-sops ys
  ⟨*proof*⟩

lemma write-sops-append: write-sops (xs@ys) = write-sops xs ∪ write-sops ys
  ⟨*proof*⟩

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

lemma in-store-sops-conv:
  (sop ∈ store-sops xs) = ((∃ volatile a A L R W. Write volatile a sop A L R W ∈ set xs) ∨
                          (∃ a t cond ret A L R W. RMW a t sop cond ret A L R W ∈ set xs))
  ⟨*proof*⟩

lemma store-sops-mono: ⋀ys. set xs ⊆ set ys ⟹ store-sops xs ⊆ store-sops ys
  ⟨*proof*⟩

lemma store-sops-append: store-sops (xs@ys) = store-sops xs ∪ store-sops ys
  ⟨*proof*⟩

locale valid-write-sops =
fixes ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes valid-write-sops:
  ⋀i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
  ⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
  ⟹
  ∀ sop ∈ write-sops sb. valid-sop sop

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

locale valid-sops = valid-write-sops + valid-store-sops

The value stored in a non-volatile $\mathsf{Read_{sb}}$ 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$_{\mathsf{sb}}$ volatile a t v $\Rightarrow$ ($\neg$ volatile $\longrightarrow$ (pending-write $\vee$ a $\in$ $\mathcal{O}$) $\longrightarrow$ v = m a) $\wedge$
                        reads-consistent pending-write $\mathcal{O}$ m rs
    | Write$_{\mathsf{sb}}$ volatile a sop v A L R W $\Rightarrow$
        (if volatile then
            outstanding-refs is-volatile-Read$_{\mathsf{sb}}$ rs = {} $\wedge$
            reads-consistent True ($\mathcal{O}$ $\cup$ A $-$ R) (m(a := v)) rs
         else reads-consistent pending-write $\mathcal{O}$ (m(a := v)) rs)
    | Ghost$_{\mathsf{sb}}$ A L R W $\Rightarrow$ reads-consistent pending-write ($\mathcal{O}$ $\cup$ A $-$ R) m rs
    | - $\Rightarrow$ 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$_{\mathsf{sb}}$ volatile a t v $\Rightarrow$ (volatile $\longrightarrow$ v = m a) $\wedge$ volatile-reads-consistent m rs
    | Write$_{\mathsf{sb}}$ volatile a sop v A L R W $\Rightarrow$ volatile-reads-consistent (m(a := v)) rs
    | - $\Rightarrow$ 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$_{\mathsf{sb}}$ volatile a - v - - - - $\Rightarrow$ flush rs (m(a:=v))
      | - $\Rightarrow$ flush rs m)

lemma reads-consistent-pending-write-antimono:
  $\bigwedge\mathcal{O}$ m. reads-consistent True $\mathcal{O}$ m sb $\Longrightarrow$ reads-consistent False $\mathcal{O}$ m sb

$\langle proof \rangle$

lemma reads-consistent-owns-antimono:
$\bigwedge\mathcal{O}\ \mathcal{O}'$ pending-write m.
$\mathcal{O} \subseteq \mathcal{O}' \Longrightarrow$ reads-consistent pending-write $\mathcal{O}'$ m sb $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m sb
$\langle proof \rangle$

lemma acquired-reads-mono$'$: x $\in$ acquired-reads b xs A $\Longrightarrow$ acquired-reads b xs B = {} $\Longrightarrow$ A $\subseteq$ B $\Longrightarrow$ False
$\langle proof \rangle$

lemma reads-consistent-append:
$\bigwedge$m pending-write $\mathcal{O}$. reads-consistent pending-write $\mathcal{O}$ m (xs@ys) =
(reads-consistent pending-write $\mathcal{O}$ m xs $\wedge$
reads-consistent (pending-write $\vee$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ xs $\neq$ {})
(acquired True xs $\mathcal{O}$) (flush xs m) ys $\wedge$
(outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ xs $\neq$ {}
$\longrightarrow$ outstanding-refs is-volatile-Read$_{\mathsf{sb}}$ ys = {}))
$\langle proof \rangle$

lemma reads-consistent-mem-eq-on-non-volatile-reads:
assumes mem-eq: $\forall$ a $\in$ A. m$'$ a = m a
assumes subset: outstanding-refs (is-non-volatile-Read$_{\mathsf{sb}}$) 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 $\mathcal{O}$ m sb
shows reads-consistent pending-write $\mathcal{O}$ m$'$ sb
$\langle proof \rangle$

lemma volatile-reads-consistent-mem-eq-on-volatile-reads:
assumes mem-eq: $\forall$ a $\in$ A. m$'$ a = m a
assumes subset: outstanding-refs (is-volatile-Read$_{\mathsf{sb}}$) 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
$\langle proof \rangle$

locale valid-reads =
fixes m::memory **and** ts::($'$p, $'$p store-buffer,bool,owns,rels) thread-config list
assumes valid-reads: $\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!] \Longrightarrow$
reads-consistent False $\mathcal{O}$ m sb

lemma valid-reads-Cons: valid-reads m (t#ts) =

(let (-,-,-,sb,-,$\mathcal{O}$,-) = t in reads-consistent False $\mathcal{O}$ m sb $\wedge$ valid-reads m ts)
$\langle proof \rangle$

$\mathsf{Read_{sb}s}$ and writes have in the store-buffer have to conform to the valuation of temporaries.**context** program
begin
fun history-consistent:: tmps $\Rightarrow$ $'$p $\Rightarrow$ $'$p store-buffer $\Rightarrow$ bool
where
history-consistent j p [] = True
| history-consistent j p (r#rs) =
(case r of
$\mathrm{Read_{sb}}$ vol a t v $\Rightarrow$
(case j t of Some v$'$ $\Rightarrow$ v=v$'$ $\wedge$ history-consistent j p rs | - $\Rightarrow$ False)
| $\mathrm{Write_{sb}}$ vol a (D,f) v - - - - $\Rightarrow$
D $\subseteq$ dom j $\wedge$ f j = v $\wedge$ D $\cap$ read-tmps rs = {} $\wedge$ history-consistent j p rs
| $\mathrm{Prog_{sb}}$ p_1 p_2 is $\Rightarrow$ p_1=p $\wedge$
j|\`(dom j $-$ read-tmps rs)$\vdash$ p_1 $\rightarrow_{p}$ (p_2,is) $\wedge$
history-consistent j p_2 rs
| - $\Rightarrow$ history-consistent j p rs)
end

fun hd-prog:: $'$p $\Rightarrow$ $'$p store-buffer $\Rightarrow$ $'$p
where
hd-prog p [] = p
| hd-prog p (i#is) = (case i of
$\mathrm{Prog_{sb}}$ p$'$ - - $\Rightarrow$ p$'$
| - $\Rightarrow$ hd-prog p is)

fun last-prog:: $'$p $\Rightarrow$ $'$p store-buffer $\Rightarrow$ $'$p
where
last-prog p [] = p
| last-prog p (i#is) = (case i of
$\mathrm{Prog_{sb}}$ - p$'$ - $\Rightarrow$ last-prog p$'$ is
| - $\Rightarrow$ last-prog p is)

locale valid-history = program +
constrains
program-step :: tmps $\Rightarrow$ $'$p $\Rightarrow$ $'$p $\times$ instrs $\Rightarrow$ bool
fixes ts::($'$p,$'$p store-buffer,bool,owns,rels) thread-config list
assumes valid-history: $\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$ $\Longrightarrow$
program.history-consistent program-step j (hd-prog p sb) sb

fun data-dependency-consistent-instrs:: addr set $\Rightarrow$ instrs $\Rightarrow$ bool
where
data-dependency-consistent-instrs T [] = True
| data-dependency-consistent-instrs T (i#is) =
(case i of
Write volatile a (D,f) - - - - $\Rightarrow$ D $\subseteq$ T $\wedge$ D $\cap$ load-tmps is = {} $\wedge$ data-dependency-consistent-instrs T is

| RMW a t (D,f) cond ret - - - - ⇒ D ⊆ T ∧ D ∩ load-tmps is = {} ∧ data-dependency-consistent-instrs (insert t T) is
| Read - - t ⇒ data-dependency-consistent-instrs (insert t T) is
| - ⇒ data-dependency-consistent-instrs T is)

lemma data-dependency-consistent-mono:
⋀ T T′. ⟦data-dependency-consistent-instrs T is; T ⊆ T′⟧ ⟹ data-dependency-consistent-instrs T′ is
⟨*proof*⟩

lemma data-dependency-consistent-instrs-append:
⋀ys T . data-dependency-consistent-instrs T (xs@ys) =
(data-dependency-consistent-instrs T xs ∧
data-dependency-consistent-instrs (T ∪ load-tmps xs) ys ∧
load-tmps ys ∩ ⋃(fst ‘ store-sops xs) = {})
⟨*proof*⟩

locale valid-data-dependency =
fixes ts::(′p,′p store-buffer,bool,owns,rels) thread-config list
assumes data-dependency-consistent-instrs:
⋀i p is 𝒪 𝒟 j sb.
⟦i < length ts; ts!i = (p,is,j,sb,𝒟,𝒪,ℛ) ⟧ ⟹
data-dependency-consistent-instrs (dom j) is
assumes load-tmps-write-tmps-distinct:
⋀i p is 𝒪 𝒟 j sb.
⟦i < length ts; ts!i = (p,is,j,sb,𝒟,𝒪,ℛ) ⟧ ⟹
load-tmps is ∩ ⋃(fst ‘ write-sops sb) = {}

locale load-tmps-fresh =
fixes ts::(′p, ′p store-buffer,bool,owns,rels) thread-config list
assumes load-tmps-fresh:
⋀i p is 𝒪 𝒟 j sb.
⟦i < length ts; ts!i = (p,is,j,sb,𝒟,𝒪,ℛ) ⟧ ⟹
load-tmps is ∩ dom j = {}

fun acquired-by-instrs :: instrs ⇒ addr set ⇒ addr set
where
acquired-by-instrs [] A = A
| acquired-by-instrs (i#is) A =
(case i of
Read - - - ⇒ acquired-by-instrs is A
| Write volatile - - A′ L R W ⇒ acquired-by-instrs is (if volatile then (A ∪ A′ − R) else A)
| RMW a t sop cond ret A′ L R W ⇒ acquired-by-instrs is {}
| Fence ⇒ acquired-by-instrs is {}
| Ghost A′ L R W ⇒ acquired-by-instrs is (A ∪ A′ − R))

fun acquired-loads :: bool ⇒ instrs ⇒ addr set ⇒ addr set
where

acquired-loads pending-write [] A = {}
| acquired-loads pending-write (i#is) A =
(case i of
Read volatile a - ⇒ (if pending-write ∧ ¬ volatile ∧ a ∈ A
then insert a (acquired-loads pending-write is A)
else acquired-loads pending-write is A)
| Write volatile - - A′ L R W ⇒ (if volatile then acquired-loads True is (if pending-write then (A ∪ A′ − R) else {})
else acquired-loads pending-write is A)
| RMW a t sop cond ret A′ L R W ⇒ acquired-loads pending-write is {}
| Fence ⇒ acquired-loads pending-write is {}
| Ghost A′ L R W ⇒ acquired-loads pending-write is (A ∪ A′ − R))

lemma acquired-by-instrs-mono:
$\bigwedge$ A B. A ⊆ B ⟹ acquired-by-instrs is A ⊆ 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,′p store-buffer,bool,owns,rels) thread-config list
assumes clean-no-outstanding-volatile-$\text{Write}_{\mathsf{sb}}$:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);¬ $\mathcal{D}$⟧ ⟹
(outstanding-refs is-volatile-$\text{Write}_{\mathsf{sb}}$ sb = {})

fun prog-instrs:: ′p store-buffer ⇒ instrs
where
prog-instrs [] = []
|prog-instrs (i#is) = (case i of
$\text{Prog}_{\mathsf{sb}}$ - - is′ ⇒ is′ @ prog-instrs is
| - ⇒ prog-instrs is)

fun instrs:: ′p store-buffer ⇒ instrs
where
instrs [] = []
| instrs (i#is) = (case i of
$\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W ⇒ Write volatile a sop A L R W# instrs is
| $\text{Read}_{\mathsf{sb}}$ volatile a t v ⇒ Read volatile a t # instrs is
| $\text{Ghost}_{\mathsf{sb}}$ A L R W ⇒ Ghost A L R W# instrs is
| - ⇒ instrs is)

locale causal-program-history =
fixes is_{sb} **and** sb
assumes causal-program-history:

$\bigwedge$sb$_1$ sb$_2$. sb=sb$_1$@sb$_2$ $\Longrightarrow$ $\exists$is. instrs sb$_2$ @ is$_{sb}$ = is @ prog-instrs sb$_2$

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

lemma causal-program-history-suffix:
causal-program-history is$_{sb}$ (sb@sb$'$) $\Longrightarrow$ 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:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟧ $\Longrightarrow$
causal-program-history is sb

assumes valid-last-prog:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟧ $\Longrightarrow$
last-prog p sb = p

lemma (**in** valid-program-history) valid-program-history-nth-update:
⟦i < length ts; causal-program-history is sb; last-prog p sb = p⟧
$\Longrightarrow$
valid-program-history (ts [i:=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

lemma (**in** outstanding-non-volatile-refs-owned-or-read-only)
outstanding-non-volatile-refs-owned-instructions-read-value-independent:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟧ $\Longrightarrow$
outstanding-non-volatile-refs-owned-or-read-only $\mathcal{S}$ (ts[i := (p$'$,is$'$,j$'$,sb,$\mathcal{D}'$,$\mathcal{O}$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma (**in** outstanding-non-volatile-refs-owned-or-read-only)
outstanding-non-volatile-refs-owned-or-read-only-nth-update:
$\bigwedge$i is $\mathcal{O}$ $\mathcal{D}$ $\mathcal{R}$ j sb.
⟦i < length ts; non-volatile-owned-or-read-only False $\mathcal{S}$ $\mathcal{O}$ sb⟧ $\Longrightarrow$
outstanding-non-volatile-refs-owned-or-read-only $\mathcal{S}$ (ts[i := (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

lemma (**in** outstanding-volatile-writes-unowned-by-others)
outstanding-volatile-writes-unowned-by-others-instructions-read-value-independent:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟧ $\Longrightarrow$
outstanding-volatile-writes-unowned-by-others (ts[i := (p$'$,is$'$,j$'$,sb,$\mathcal{D}'$,$\mathcal{O}$,$\mathcal{R}'$)])
⟨*proof*⟩

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

$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!] \Longrightarrow$
read-only-reads-unowned (ts[i := (p′,is′,j′,sb,$\mathcal{D}'$,$\mathcal{O}$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma Write$_{\mathsf{sb}}$-in-outstanding-refs:
Write$_{\mathsf{sb}}$ True a sop v A L R W ∈ set xs $\Longrightarrow$ a ∈ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ xs
⟨*proof*⟩

lemma (**in** outstanding-volatile-writes-unowned-by-others)
outstanding-volatile-writes-unowned-by-others-store-buffer:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb′ ⊆ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb;
all-acquired sb′ ⊆ all-acquired sb$]\!] \Longrightarrow$
outstanding-volatile-writes-unowned-by-others (ts[i := (p′,is′,j′,sb′,$\mathcal{D}'$,$\mathcal{O}$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma (**in** ownership-distinct)
ownership-distinct-instructions-read-value-store-buffer-independent:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
all-acquired sb′ ⊆ all-acquired sb$]\!] \Longrightarrow$
ownership-distinct (ts[i := (p′,is′,j′,sb′,$\mathcal{D}'$,$\mathcal{O}$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma (**in** ownership-distinct)
ownership-distinct-nth-update:
$\bigwedge$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ xs sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
∀j < length ts. i≠j ⟶ (let (p_j,is_j,j_j,sb_j,$\mathcal{D}_j$,$\mathcal{O}_j$,$\mathcal{R}_j$) = ts!j
in ($\mathcal{O}'$ ∪ all-acquired sb′) ∩ ($\mathcal{O}_j$ ∪ all-acquired sb_j) ={}) $]\!] \Longrightarrow$
ownership-distinct (ts[i := (p′,is′,j′,sb′,$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma (**in** valid-write-sops) valid-write-sops-nth-update:
$[\![$i < length ts; ∀sop ∈ write-sops sb. valid-sop sop$]\!] \Longrightarrow$
valid-write-sops (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

lemma (**in** valid-store-sops) valid-store-sops-nth-update:
$[\![$i < length ts; ∀sop ∈ store-sops is. valid-sop sop$]\!] \Longrightarrow$
valid-store-sops (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

lemma (**in** valid-sops) valid-sops-nth-update:
$[\![$i < length ts; $\forall$ sop $\in$ write-sops sb. valid-sop sop;
$\forall$ sop $\in$ store-sops is. valid-sop sop$]\!] \Longrightarrow$
valid-sops (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
$\langle$*proof*$\rangle$

lemma (**in** valid-data-dependency) valid-data-dependency-nth-update:
$[\![$i < length ts; data-dependency-consistent-instrs (dom j) is;
load-tmps is $\cap \bigcup$(fst ‘ write-sops sb) = {}$]\!] \Longrightarrow$
valid-data-dependency (ts[i := (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
$\langle$*proof*$\rangle$

lemma (**in** enough-flushs) enough-flushs-nth-update:
$[\![$i < length ts;
$\neg\ \mathcal{D} \longrightarrow$ (outstanding-refs is-volatile-$\text{Write}_{\mathsf{sb}}$ sb = {})
$]\!] \Longrightarrow$
enough-flushs (ts[i := (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])

$\langle$*proof*$\rangle$

lemma (**in** outstanding-non-volatile-writes-unshared)
outstanding-non-volatile-writes-unshared-nth-update:
$[\![$i < length ts; non-volatile-writes-unshared $\mathcal{S}$ sb$]\!] \Longrightarrow$
outstanding-non-volatile-writes-unshared $\mathcal{S}$ (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
$\langle$*proof*$\rangle$

lemma (**in** sharing-consis)
sharing-consis-nth-update:
$[\![$i < length ts; sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb$]\!] \Longrightarrow$
sharing-consis $\mathcal{S}$ (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
$\langle$*proof*$\rangle$

lemma (**in** no-outstanding-write-to-read-only-memory)
no-outstanding-write-to-read-only-memory-nth-update:
$[\![$i < length ts; no-write-to-read-only-memory $\mathcal{S}$ sb$]\!] \Longrightarrow$
no-outstanding-write-to-read-only-memory $\mathcal{S}$ (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
$\langle$*proof*$\rangle$

lemma in-Union-image-nth-conv: a $\in \bigcup$ (f ‘ set xs) $\Longrightarrow \exists$ i. i < length xs $\wedge$ a $\in$ f (xs!i)
$\langle$*proof*$\rangle$

lemma in-Inter-image-nth-conv: a $\in \bigcap$ (f ‘ set xs) = ($\forall$ i < length xs. a $\in$ f (xs!i))
$\langle$*proof*$\rangle$

lemma release-ownership-nth-update:
  assumes R-subset: R ⊆ $\mathcal{O}$
  shows ⋀i. ⟦i < length ts; ts!i = (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
        ownership-distinct ts⟧
   ⟹ ⋃ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ' set (ts[i:=(p′,is′,xs′,sb′,$\mathcal{D}$′,$\mathcal{O}$ − R,$\mathcal{R}$′)]))
      = ((⋃ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ' set ts)) − R)
⟨*proof*⟩

lemma acquire-ownership-nth-update:
  shows ⋀i. ⟦i < length ts; ts!i = (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
   ⟹ ⋃ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ' set (ts[i:=(p′,is′,xs′,sb′,$\mathcal{D}$′,$\mathcal{O}$ ∪ A,$\mathcal{R}$′)]))
      = ((⋃ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ' set ts)) ∪ A)
⟨*proof*⟩

lemma acquire-release-ownership-nth-update:
  assumes R-subset: R ⊆ $\mathcal{O}$
  shows ⋀i. ⟦i < length ts; ts!i = (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
        ownership-distinct ts⟧
   ⟹ ⋃ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ' set (ts[i:=(p′,is′,xs′,sb′,$\mathcal{D}$′,$\mathcal{O}$ ∪ A − R,$\mathcal{R}$′)]))
      = ((⋃ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ' set ts)) ∪ A − R)
⟨*proof*⟩

lemma (**in** valid-history) valid-history-nth-update:
    ⟦i < length ts; history-consistent j (hd-prog p sb) sb ⟧ ⟹
      valid-history program-step (ts[i := (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
  ⟨*proof*⟩

lemma (**in** valid-reads) valid-reads-nth-update:
    ⟦i < length ts; reads-consistent False $\mathcal{O}$ m sb ⟧ ⟹
      valid-reads m (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
  ⟨*proof*⟩

lemma (**in** load-tmps-distinct) load-tmps-distinct-nth-update:
    ⟦i < length ts; distinct-load-tmps is⟧ ⟹
      load-tmps-distinct (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
  ⟨*proof*⟩

lemma (**in** read-tmps-distinct) read-tmps-distinct-nth-update:
    ⟦i < length ts; distinct-read-tmps sb⟧ ⟹
      read-tmps-distinct (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
  ⟨*proof*⟩

lemma (**in** load-tmps-read-tmps-distinct) load-tmps-read-tmps-distinct-nth-update:
    ⟦i < length ts; load-tmps is ∩ read-tmps sb = {}⟧ ⟹
      load-tmps-read-tmps-distinct (ts[i := (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
  ⟨*proof*⟩

lemma (**in** load-tmps-fresh) load-tmps-fresh-nth-update:

$[\![$i < length ts;
load-tmps is ∩ dom j = {}$]\!]$ $\Longrightarrow$
load-tmps-fresh (ts[i := (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

fun flush-all-until-volatile-write::
('p,'p store-buffer,'dirty,'owns,'rels) thread-config list ⇒ memory ⇒ memory
where
flush-all-until-volatile-write [] m = m
| flush-all-until-volatile-write ((-, -, -, sb,-, -)#ts) m =
flush-all-until-volatile-write ts (flush (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) m)

fun share-all-until-volatile-write::
('p,'p store-buffer,'dirty,'owns,'rels) thread-config list ⇒ shared ⇒ shared
where
share-all-until-volatile-write [] S = S
| share-all-until-volatile-write ((-, -, -, sb,-,-)#ts) S =
share-all-until-volatile-write ts (share (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) S)

lemma takeWhile-dropWhile-real-prefix:
$[\![$x ∈ set xs; ¬ P x$]\!]$ $\Longrightarrow$ ∃ y ys. xs=takeWhile P xs @ y#ys ∧ ¬ P y ∧ dropWhile P xs = y#ys
⟨*proof*⟩

lemma buffered-val-witness: buffered-val sb a = Some v $\Longrightarrow$
∃ volatile sop A L R W. Write$_{\mathsf{sb}}$ volatile a sop v A L R W ∈ set sb
⟨*proof*⟩

lemma flush-append-Read$_{\mathsf{sb}}$:
⋀m. (flush (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Read$_{\mathsf{sb}}$ volatile a t v])) m)
= flush (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) m
⟨*proof*⟩

lemma flush-append-write:
⋀m. (flush (sb @ [Write$_{\mathsf{sb}}$ volatile a sop v A L R W]) m) = (flush sb m) (a:=v)
⟨*proof*⟩

lemma flush-append-Prog$_{\mathsf{sb}}$:
⋀m. (flush (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Prog$_{\mathsf{sb}}$ p_1 p_2 mis])) m) =
(flush (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) m)
⟨*proof*⟩

lemma flush-append-Ghost$_{\mathsf{sb}}$:
$\bigwedge$m. (flush (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Ghost$_{\mathsf{sb}}$ A L R W])) m) =
(flush (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) m)
$\langle$*proof*$\rangle$

lemma share-append: $\bigwedge$S. share (xs@ys) S = share ys (share xs S)
$\langle$*proof*$\rangle$

lemma share-append-Read$_{\mathsf{sb}}$:
$\bigwedge$S. (share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Read$_{\mathsf{sb}}$ volatile a t v])) S)
= share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) S
$\langle$*proof*$\rangle$

lemma share-append-Write$_{\mathsf{sb}}$:
$\bigwedge$S. (share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Write$_{\mathsf{sb}}$ volatile a sop v A L R W])) S)
= share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) S
$\langle$*proof*$\rangle$

lemma share-append-Prog$_{\mathsf{sb}}$:
$\bigwedge$S. (share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Prog$_{\mathsf{sb}}$ p$_1$ p$_2$ mis])) S) =
(share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) S)
$\langle$*proof*$\rangle$

lemma in-acquired-no-pending-write-outstanding-write:
a $\in$ acquired False sb A $\Longrightarrow$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb $\neq$ {}
$\langle$*proof*$\rangle$

lemma flush-buffered-val-conv:
$\bigwedge$m. flush sb m a = (case buffered-val sb a of None $\Rightarrow$ m a | Some v $\Rightarrow$ v)
$\langle$*proof*$\rangle$

lemma reads-consistent-unbuffered-snoc:
$\bigwedge$m. buffered-val sb a = None $\Longrightarrow$ m a = v $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$
volatile $\longrightarrow$
outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb = {}
$\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m (sb @ [Read$_{\mathsf{sb}}$ volatile a t v])
$\langle$*proof*$\rangle$

lemma reads-consistent-buffered-snoc:
$\bigwedge$m. buffered-val sb a = Some v $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$
volatile $\longrightarrow$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb = {}
$\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m (sb @ [Read$_{\mathsf{sb}}$ volatile a t v])
$\langle$*proof*$\rangle$

lemma reads-consistent-snoc-Write$_{\mathsf{sb}}$:
$\bigwedge$m. reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$

reads-consistent pending-write $\mathcal{O}$ m (sb @ [$\text{Write}_{\mathsf{sb}}$ volatile a sop v A L R W])
⟨*proof*⟩

lemma reads-consistent-snoc-$\text{Prog}_{\mathsf{sb}}$:
$\bigwedge$m. reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m (sb @ [$\text{Prog}_{\mathsf{sb}}$ p_1 p_2 mis])
⟨*proof*⟩

lemma reads-consistent-snoc-$\text{Ghost}_{\mathsf{sb}}$:
$\bigwedge$m. reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m (sb @ [$\text{Ghost}_{\mathsf{sb}}$ A L R W])
⟨*proof*⟩

lemma restrict-map-id [simp]:m |‘ dom m = m
⟨*proof*⟩

lemma flush-all-until-volatile-write-Read-commute:
shows $\bigwedge$m i. ⟦i < length ls; ls!i=(p,Read volatile a t#is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
⟧
$\Longrightarrow$
flush-all-until-volatile-write
(ls[i := (p,is , j(t↦v), sb @ [$\text{Read}_{\mathsf{sb}}$ volatile a t v],$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)]) m =
flush-all-until-volatile-write ls m
⟨*proof*⟩

lemma flush-all-until-volatile-write-append-Ghost-commute:
$\bigwedge$i m. ⟦i < length ts; ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
$\Longrightarrow$ flush-all-until-volatile-write (ts[i := (p′,is′,j′, sb@[$\text{Ghost}_{\mathsf{sb}}$ A L R W], $\mathcal{D}'$, $\mathcal{O}'$,$\mathcal{R}'$)]) m
= flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma update-commute:
assumes g-unchanged: ∀ a m. a ∉ G ⟶ g m a = m a
assumes g-independent: ∀ a m. a ∈ G ⟶ g (f m) a = g m a
assumes f-unchanged: ∀ a m. a ∉ F ⟶ f m a = m a
assumes f-independent: ∀ a m. a ∈ F ⟶ f (g m) a = f m a
assumes disj: G ∩ F = {}
shows f (g m) = g (f m)
⟨*proof*⟩

lemma update-commute′:
assumes g-unchanged: ∀ a m. a ∉ G ⟶ g m a = m a
assumes g-independent: ∀ a m_1 m_2. a ∈ G ⟶ g m_1 a = g m_2 a
assumes f-unchanged: ∀ a m. a ∉ F ⟶ f m a = m a
assumes f-independent: ∀ a m_1 m_2. a ∈ F ⟶ f m_1 a = f m_2 a
assumes disj: G ∩ F = {}

shows f (g m) = g (f m)
⟨*proof*⟩

lemma flush-unchanged-addresses: $\bigwedge$m. a $\notin$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb $\Longrightarrow$ flush sb m a = m a
⟨*proof*⟩

lemma flushed-values-mem-independent:
$\bigwedge$m m′ a. a $\in$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb $\Longrightarrow$ flush sb m′ a = flush sb m a
⟨*proof*⟩

lemma flush-all-until-volatile-write-unchanged-addresses:
$\bigwedge$m. a $\notin \bigcup$((λ(-,-,-,sb,-,-,-). outstanding-refs is-Write$_{\mathsf{sb}}$
(takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)) ‘ set ls) $\Longrightarrow$
flush-all-until-volatile-write ls m a = m a
⟨*proof*⟩

lemma notin-outstanding-non-volatile-takeWhile-lem:
a $\notin$ outstanding-refs (Not $\circ$ is-volatile) sb
$\Longrightarrow$
a $\notin$ outstanding-refs is-Write$_{\mathsf{sb}}$ (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
⟨*proof*⟩

lemma notin-outstanding-non-volatile-takeWhile-lem′:
a $\notin$ outstanding-refs is-non-volatile-Write$_{\mathsf{sb}}$ sb
$\Longrightarrow$
a $\notin$ outstanding-refs is-Write$_{\mathsf{sb}}$ (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
⟨*proof*⟩

lemma notin-outstanding-non-volatile-takeWhile-Un-lem′:
a $\notin \bigcup$ ((λ(-,-,-,sb,-,-,-). outstanding-refs (Not $\circ$ is-volatile) sb) ‘ set ls)
$\Longrightarrow$ a $\notin \bigcup$ ((λ(-,-,-,sb,-,-,-). outstanding-refs is-Write$_{\mathsf{sb}}$
(takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)) ‘ set ls)
⟨*proof*⟩

lemma flush-all-until-volatile-write-unchanged-addresses′:
assumes notin: a $\notin \bigcup$ ((λ(-,-,-,sb,-,-,-). outstanding-refs (Not $\circ$ is-volatile) sb) ‘ set ls)
shows flush-all-until-volatile-write ls m a = m a
⟨*proof*⟩

lemma flush-all-until-volatile-wirte-mem-independent:
$\bigwedge$m m′. a $\in \bigcup$ ((λ(-,-,-,sb,-,-,-). outstanding-refs is-Write$_{\mathsf{sb}}$
(takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)) ‘ set ls) $\Longrightarrow$
flush-all-until-volatile-write ls m′ a = flush-all-until-volatile-write ls m a
⟨*proof*⟩

lemma flush-all-until-volatile-write-buffered-val-conv:
assumes no-volatile-Write$_{\mathsf{sb}}$: outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb = {}
shows$\bigwedge$m i. $[\![$i < length ls; ls!i = (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);

$\forall$ j < length ls. i $\neq$ j $\longrightarrow$
(let (-,-,-,sb_j,-,-,-) = ls!j
in a $\notin$ outstanding-refs is-non-volatile-$Write_{sb}$ (takeWhile (Not $\circ$ is-volatile-$Write_{sb}$) sb_j)) $\rrbracket \Longrightarrow$
flush-all-until-volatile-write ls m a =
(case buffered-val sb a of None $\Rightarrow$ m a | Some v $\Rightarrow$ v)
$\langle proof \rangle$

context program
begin

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

term x $\Rightarrow_{sb}$ Y

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

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

abbreviation sbh-concurrent-step ::
('p,'p store-buffer,bool,owns,rels,shared) global-config $\Rightarrow$ ('p,'p store-buffer,bool,owns,rels,shared) global-config $\Rightarrow$ bool
(‹- $\Rightarrow_{sbh}$ -› [60,60] 100)
where
sbh-concurrent-step $\equiv$
computation.concurrent-step sbh-memop-step flush-step program-step
(λp p' is sb. sb @ [$Prog_{sb}$ p p' is])

term x $\Rightarrow_{sbh}$ Y

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

term x $\Rightarrow_{\mathsf{sbh}}{}^*$ Y
end

lemma instrs-append-Read$_{\mathsf{sb}}$:
instrs (sb@[Read$_{\mathsf{sb}}$ volatile a t v]) = instrs sb @ [Read volatile a t]
⟨*proof*⟩

lemma instrs-append-Write$_{\mathsf{sb}}$:
instrs (sb@[Write$_{\mathsf{sb}}$ volatile a sop v A L R W]) = instrs sb @ [Write volatile a sop A L R W]
⟨*proof*⟩

lemma instrs-append-Ghost$_{\mathsf{sb}}$:
instrs (sb@[Ghost$_{\mathsf{sb}}$ A L R W]) = instrs sb @ [Ghost A L R W]
⟨*proof*⟩

lemma prog-instrs-append-Ghost$_{\mathsf{sb}}$:
prog-instrs (sb@[Ghost$_{\mathsf{sb}}$ A L R W]) = prog-instrs sb
⟨*proof*⟩

lemma prog-instrs-append-Read$_{\mathsf{sb}}$:
prog-instrs (sb@[Read$_{\mathsf{sb}}$ volatile a t v]) = prog-instrs sb
⟨*proof*⟩

lemma prog-instrs-append-Write$_{\mathsf{sb}}$:
prog-instrs (sb@[Write$_{\mathsf{sb}}$ volatile a sop v A L R W]) = prog-instrs sb
⟨*proof*⟩

lemma hd-prog-append-Read$_{\mathsf{sb}}$:
hd-prog p (sb@[Read$_{\mathsf{sb}}$ volatile a t v]) = hd-prog p sb
⟨*proof*⟩

lemma hd-prog-append-Write$_{\mathsf{sb}}$:
hd-prog p (sb@[Write$_{\mathsf{sb}}$ volatile a sop v A L R W]) = hd-prog p sb
⟨*proof*⟩

lemma flush-update-other: $\bigwedge$m. a $\notin$ outstanding-refs (Not $\circ$ is-volatile) sb $\Longrightarrow$
outstanding-refs (is-volatile-Write$_{\mathsf{sb}}$) sb = {} $\Longrightarrow$
flush sb (m(a:=v)) = (flush sb m)(a := v)
⟨*proof*⟩

lemma flush-update-other′: $\bigwedge$m. a $\notin$ outstanding-refs (is-non-volatile-Write$_{\mathsf{sb}}$) sb $\Longrightarrow$
outstanding-refs (is-volatile-Write$_{\mathsf{sb}}$) sb = {} $\Longrightarrow$
flush sb (m(a:=v)) = (flush sb m)(a := v)
⟨*proof*⟩

lemma flush-update-other″: $\bigwedge$m. a $\notin$ outstanding-refs (is-non-volatile-Write$_{\mathsf{sb}}$) sb $\Longrightarrow$
a $\notin$ outstanding-refs (is-volatile-Write$_{\mathsf{sb}}$) sb $\Longrightarrow$
flush sb (m(a:=v)) = (flush sb m)(a := v)

⟨*proof*⟩

lemma flush-all-until-volatile-write-update-other:
$\bigwedge$m. $\forall$ j < length ts.
(let (-,-,-,sb_j,-,-,-) = ts!j
in a $\notin$ outstanding-refs is-non-volatile-$Write_{sb}$ (takeWhile (Not $\circ$ is-volatile-$Write_{sb}$) sb_j))
$\Longrightarrow$
flush-all-until-volatile-write ts (m(a := v)) =
(flush-all-until-volatile-write ts m)(a := v)
⟨*proof*⟩

lemma flush-all-until-volatile-write-append-non-volatile-write-commute:
assumes no-volatile-$Write_{sb}$: outstanding-refs is-volatile-$Write_{sb}$ sb = {}
shows $\bigwedge$m i. $[\![$i < length ts; ts!i = (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
$\forall$ j < length ts. i $\neq$ j $\longrightarrow$
(let (-,-,-,sb_j,-,-,-) = ts!j
in a $\notin$ outstanding-refs is-non-volatile-$Write_{sb}$ (takeWhile (Not $\circ$ is-volatile-$Write_{sb}$) sb_j))$]\!]$
$\Longrightarrow$ flush-all-until-volatile-write (ts[i := (p′,is′, xs, sb @ [$Write_{sb}$ False a sop v A L R W],$\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) m =
(flush-all-until-volatile-write ts m)(a := v)
⟨*proof*⟩

lemma flush-all-until-volatile-write-append-unflushed:
assumes volatile-$Write_{sb}$: $\neg$ outstanding-refs is-volatile-$Write_{sb}$ sb = {}
shows $\bigwedge$m i. $[\![$i < length ts; ts!i = (p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!]$
$\Longrightarrow$ flush-all-until-volatile-write (ts[i := (p′,is′, xs, sb @ sbx,$\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) m =
(flush-all-until-volatile-write ts m)
⟨*proof*⟩

lemma flush-all-until-volatile-nth-update-unused:
shows $\bigwedge$m i. $[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!]$
$\Longrightarrow$ flush-all-until-volatile-write (ts[i := (p′,is′,j′, sb, $\mathcal{D}'$, $\mathcal{O}'$,$\mathcal{R}'$)]) m =
(flush-all-until-volatile-write ts m)
⟨*proof*⟩

lemma flush-all-until-volatile-write-append-volatile-write-commute:
assumes no-volatile-$Write_{sb}$: outstanding-refs is-volatile-$Write_{sb}$ sb = {}
shows $\bigwedge$m i. $[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!]$ $\Longrightarrow$
flush-all-until-volatile-write
(ts[i := (p′,is′, j, sb @ [$Write_{sb}$ True a sop v A L R W],$\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) m
= flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma reads-consistent-update:
$\bigwedge$pending-write $\mathcal{O}$ m. reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$
a $\notin$ outstanding-refs (Not $\circ$ is-volatile) sb $\Longrightarrow$
reads-consistent pending-write $\mathcal{O}$ (m(a := v)) sb
⟨*proof*⟩

lemma (**in** program) history-consistent-hd-prog: $\bigwedge$p. history-consistent j p′ xs
$\Longrightarrow$ history-consistent j (hd-prog p xs) xs
⟨*proof*⟩

locale valid-program = program +
  fixes valid-prog
  assumes valid-prog-inv: ⟦j⊢p $\rightarrow_{\mathsf{p}}$ (p′,is′); valid-prog p⟧ $\Longrightarrow$ valid-prog p′

lemma (**in** valid-program) history-consistent-appendD:
  $\bigwedge$j ys p. ∀sop ∈ write-sops xs. valid-sop sop $\Longrightarrow$
        read-tmps xs ∩ read-tmps ys = {} $\Longrightarrow$
      history-consistent j p (xs@ys) $\Longrightarrow$
      (history-consistent (j|‘ (dom j − read-tmps ys)) p xs ∧
       history-consistent j (last-prog p xs) ys ∧
       read-tmps ys ∩ $\bigcup$(fst ‘ write-sops xs) = {})
⟨*proof*⟩

lemma (**in** valid-program) history-consistent-appendI:
  $\bigwedge$j ys p. ∀sop ∈ write-sops xs. valid-sop sop $\Longrightarrow$
  history-consistent (j|‘ (dom j − read-tmps ys)) p xs $\Longrightarrow$
 history-consistent j (last-prog p xs) ys $\Longrightarrow$
 read-tmps ys ∩ $\bigcup$(fst ‘ write-sops xs) = {} $\Longrightarrow$ valid-prog p $\Longrightarrow$
      history-consistent j p (xs@ys)
⟨*proof*⟩

lemma (**in** valid-program) history-consistent-append-conv:
  $\bigwedge$j ys p. ∀sop ∈ write-sops xs. valid-sop sop $\Longrightarrow$
        read-tmps xs ∩ read-tmps ys = {} $\Longrightarrow$ valid-prog p $\Longrightarrow$
      history-consistent j p (xs@ys) =
      (history-consistent (j|‘ (dom j − read-tmps ys)) p xs ∧
       history-consistent j (last-prog p xs) ys ∧
       read-tmps ys ∩ $\bigcup$(fst ‘ write-sops xs) = {})
⟨*proof*⟩

lemma instrs-takeWhile-dropWhile-conv:
  instrs xs = instrs (takeWhile P xs) @ instrs (dropWhile P xs)
⟨*proof*⟩

lemma (**in** program) history-consistent-hd-prog-p:
  $\bigwedge$p. history-consistent j p xs $\Longrightarrow$ p = hd-prog p xs
  ⟨*proof*⟩

lemma instrs-append: $\bigwedge$ys. instrs (xs@ys) = instrs xs @ instrs ys
  ⟨*proof*⟩

lemma prog-instrs-append: $\bigwedge$ys. prog-instrs (xs@ys) = prog-instrs xs @ prog-instrs ys
  ⟨*proof*⟩

lemma prog-instrs-empty: $\forall$ r $\in$ set xs. $\neg$ is-Prog$_{sb}$ r $\Longrightarrow$ prog-instrs xs = []
⟨*proof*⟩

lemma length-dropWhile [termination-simp]: length (dropWhile P xs) $\leq$ length xs
⟨*proof*⟩

lemma prog-instrs-filter-is-Prog$_{sb}$: prog-instrs (filter (is-Prog$_{sb}$) xs) = prog-instrs xs
⟨*proof*⟩

lemma Cons-to-snoc: $\bigwedge$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$_{sb}$ p$_1$ p$_2$ 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 = hd-prog p sb ; last-prog p sb = p$_{sb}]\!] \Longrightarrow$ p = hd-prog p$_{sb}$ sb
⟨*proof*⟩

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

lemma last-prog-idem: last-prog (last-prog p sb) sb = 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-$\text{Read}_{\mathsf{sb}}$:
$\bigwedge$p. last-prog p (sb @ [$\text{Read}_{\mathsf{sb}}$ volatile a t v]) = last-prog p sb
⟨*proof*⟩

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

lemma last-prog-append-$\text{Prog}_{\mathsf{sb}}$:
$\bigwedge$x. last-prog x (sb@[$\text{Prog}_{\mathsf{sb}}$ p p′ mis]) = p′
⟨*proof*⟩

lemma hd-prog-append-$\text{Prog}_{\mathsf{sb}}$: hd-prog x (sb @ [$\text{Prog}_{\mathsf{sb}}$ p p′ mis]) = hd-prog p sb
⟨*proof*⟩

lemma hd-prog-last-prog-append-$\text{Prog}_{\mathsf{sb}}$:
$\bigwedge$p′. hd-prog p′ xs = p′ $\Longrightarrow$ last-prog p′ xs = p_1 $\Longrightarrow$
hd-prog p′ (xs @ [$\text{Prog}_{\mathsf{sb}}$ p_1 p_2 mis]) = p′
⟨*proof*⟩

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

lemma last-prog-append-$\text{Ghost}_{\mathsf{sb}}$:
$\bigwedge$p. last-prog p (sb @ [$\text{Ghost}_{\mathsf{sb}}$ A L R W]) = last-prog p sb
⟨*proof*⟩

lemma dropWhile-all-False-conv:
$\forall$x $\in$ set xs. $\neg$ P x $\Longrightarrow$ dropWhile P xs = xs
⟨*proof*⟩

lemma dropWhile-append-all-False:
$\forall$y $\in$ set ys. $\neg$ P y $\Longrightarrow$
dropWhile P (xs@ys) = dropWhile P xs @ ys
⟨*proof*⟩

lemma reads-consistent-append-first:
$\bigwedge$m ys. reads-consistent pending-write $\mathcal{O}$ m (xs @ ys) $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ m xs
⟨*proof*⟩

lemma reads-consistent-takeWhile:
assumes consis: reads-consistent pending-write $\mathcal{O}$ m sb
shows reads-consistent pending-write $\mathcal{O}$ m (takeWhile P sb)
$\langle proof \rangle$

lemma flush-flush-all-until-volatile-write-Write$_{sb}$-volatile-commute:
$\bigwedge$i m. $[\![$i < length ts; ts!i=(p,is,xs,Write$_{sb}$ True a sop v A L R W#sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
$\forall$i < length ts. ($\forall$j < length ts. i $\neq$ j $\longrightarrow$
(let (-,-,-,sb$_i$,-,-,-) = ts!i;
(-,-,-,sb$_j$,-,-,-) = ts!j
in outstanding-refs is-Write$_{sb}$ sb$_i$ $\cap$
outstanding-refs is-Write$_{sb}$ (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$) =
{}));
$\forall$j < length ts. i $\neq$ j $\longrightarrow$
(let (-,-,-,sb$_j$,-,-,-) = ts!j in a $\notin$ outstanding-refs is-Write$_{sb}$ (takeWhile (Not $\circ$
is-volatile-Write$_{sb}$) sb$_j$))$]\!]$
$\Longrightarrow$
flush (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)
((flush-all-until-volatile-write ts m)(a := v)) =
flush-all-until-volatile-write (ts[i := (p,is,xs, sb, $\mathcal{D}'$, $\mathcal{O}'$,$\mathcal{R}'$)])
(m(a := v))
$\langle proof \rangle$

lemma (**in** program)
$\bigwedge$sb$'$ p. history-consistent j (hd-prog p (sb@sb$'$)) (sb@sb$'$) $\Longrightarrow$
last-prog p (sb@sb$'$) = p $\Longrightarrow$
last-prog (hd-prog p (sb@sb$'$)) sb = hd-prog p sb$'$
$\langle proof \rangle$

lemma last-prog-to-last-prog-same: $\bigwedge$p$'$. last-prog p$'$ sb = p $\Longrightarrow$ last-prog p sb = p
$\langle proof \rangle$

lemma last-prog-hd-prog-same: $[\![$last-prog p$'$ sb = p; hd-prog p$'$ sb = p$'$$]\!]$ $\Longrightarrow$ hd-prog p
sb = p$'$
$\langle proof \rangle$

lemma last-prog-hd-prog-last-prog:
last-prog p$'$ (sb@sb$'$) = p $\Longrightarrow$ hd-prog p$'$ (sb@sb$'$) = p$'$ $\Longrightarrow$
last-prog (hd-prog p sb$'$) sb = last-prog p$'$ sb
$\langle proof \rangle$

lemma (**in** program) last-prog-hd-prog-append$'$:
$\bigwedge$sb$'$ p. history-consistent j (hd-prog p (sb@sb$'$)) (sb@sb$'$) $\Longrightarrow$

last-prog p (sb@sb′) = p ⟹
last-prog (hd-prog p sb′) sb = hd-prog p sb′
⟨*proof*⟩

lemma flush-all-until-volatile-write-Write$_{sb}$-non-volatile-commute:
⋀i m. ⟦i < length ts; ts!i=(p,is,xs,Write$_{sb}$ False a sop v A L R W#sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
∀ i < length ts. (∀ j < length ts. i ≠ j ⟶
(let (-,-,-,sb$_i$,-,-,-) = ts!i;
(-,-,-,sb$_j$,-,-,-) = ts!j
in outstanding-refs is-Write$_{sb}$ sb$_i$ ∩
outstanding-refs is-Write$_{sb}$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb$_j$) =
{}));
∀ j < length ts. i ≠ j ⟶
(let (-,-,-,sb$_j$,-,-,-) = ts!j in a ∉ outstanding-refs is-Write$_{sb}$ (takeWhile (Not ∘
is-volatile-Write$_{sb}$) sb$_j$))⟧
⟹ flush-all-until-volatile-write (ts[i := (p,is, xs, sb,$\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)])(m(a := v)) =
flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma (**in** program) history-consistent-access-last-read′:
⋀p. history-consistent j p (sb @ [Read$_{sb}$ volatile a t v]) ⟹
j t = Some v
⟨*proof*⟩

lemma (**in** program) history-consistent-access-last-read:
history-consistent j p (rev (Read$_{sb}$ volatile a t v # sb)) ⟹ j t = Some v
⟨*proof*⟩

lemma flush-all-until-volatile-write-Read$_{sb}$-commute:
⋀i m. ⟦i < length ts; ts!i=(p,is,j,Read$_{sb}$ volatile a t v#sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
⟹ flush-all-until-volatile-write (ts[i := (p,is,j, sb, $\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) m
= flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma flush-all-until-volatile-write-Ghost$_{sb}$-commute:
⋀i m. ⟦i < length ts; ts!i=(p,is,j,Ghost$_{sb}$ A L R W#sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
⟹ flush-all-until-volatile-write (ts[i := (p′,is′,j′, sb, $\mathcal{D}'$, $\mathcal{O}'$,$\mathcal{R}'$)]) m
= flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma flush-all-until-volatile-write-Prog$_{sb}$-commute:
⋀i m. ⟦i < length ts; ts!i=(p,is,j,Prog$_{sb}$ p_1 p_2 mis#sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
⟹ flush-all-until-volatile-write (ts[i := (p,is, j, sb,$\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) m
= flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma flush-all-until-volatile-write-append-Prog$_{sb}$-commute:
⋀i m. ⟦i < length ts; ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧

$\Longrightarrow$ flush-all-until-volatile-write (ts[i := (p_2,is@mis, j, sb@[$\text{Prog}_{\mathsf{sb}}$ p_1 p_2 mis],$\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) m
= flush-all-until-volatile-write ts m
⟨*proof*⟩

lemma (**in** program) history-consistent-append-$\text{Prog}_{\mathsf{sb}}$:
assumes step: j$\vdash$ p $\rightarrow_{\mathsf{p}}$ (p$'$, mis)
shows history-consistent j (hd-prog p xs) xs $\Longrightarrow$ last-prog p xs = p $\Longrightarrow$
history-consistent j (hd-prog p$'$ (xs@[$\text{Prog}_{\mathsf{sb}}$ p p$'$ mis])) (xs@[$\text{Prog}_{\mathsf{sb}}$ p p$'$ mis])
⟨*proof*⟩

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

lemma augment-rels-shared-exchange: $\forall$a $\in$ R. (a $\in$ S$'$) = (a $\in$ S) $\Longrightarrow$ augment-rels S R $\mathcal{R}$ = augment-rels S$'$ R $\mathcal{R}$
⟨*proof*⟩

lemma sharing-consistent-shared-exchange:
assumes shared-eq: $\forall$a $\in$ all-acquired sb. $\mathcal{S}'$ a = $\mathcal{S}$ a
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows sharing-consistent $\mathcal{S}'$ $\mathcal{O}$ sb
⟨*proof*⟩

lemma release-shared-exchange:
assumes shared-eq: $\forall$a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb. $\mathcal{S}'$ a = $\mathcal{S}$ a
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows release sb (dom $\mathcal{S}'$) $\mathcal{R}$ = release sb (dom $\mathcal{S}$) $\mathcal{R}$
⟨*proof*⟩

lemma release-append:

$\bigwedge \mathcal{S}\ \mathcal{R}$. release (sb@xs) (dom $\mathcal{S}$) $\mathcal{R}$ = release xs (dom (share sb $\mathcal{S}$)) (release sb (dom ($\mathcal{S}$)) $\mathcal{R}$)
⟨*proof*⟩

locale xvalid-program = valid-program +
fixes valid
assumes valid-implies-valid-prog:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); valid ts⟧ ⟹ valid-prog p

assumes valid-implies-valid-prog-hd:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); valid ts⟧ ⟹ valid-prog (hd-prog p sb)
assumes distinct-load-tmps-prog-step:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); j⊢p $\rightarrow_{\mathsf{p}}$ (p′,is′); valid ts⟧
⟹
distinct-load-tmps is′ ∧
(load-tmps is′ ∩ load-tmps is = {}) ∧
(load-tmps is′ ∩ read-tmps sb) = {}

assumes valid-data-dependency-prog-step:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); j⊢p $\rightarrow_{\mathsf{p}}$ (p′,is′); valid ts⟧
⟹
data-dependency-consistent-instrs (dom j ∪ load-tmps is) is′ ∧
load-tmps is′ ∩ ⋃(fst ‘ store-sops is) = {} ∧
load-tmps is′ ∩ ⋃(fst ‘ write-sops sb) = {}

assumes load-tmps-fresh-prog-step:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); j⊢p $\rightarrow_{\mathsf{p}}$ (p′,is′); valid ts⟧
⟹
load-tmps is′ ∩ dom j = {}

assumes valid-sops-prog-step:
⟦j⊢p $\rightarrow_{\mathsf{p}}$ (p′,is′); valid-prog p⟧⟹ ∀sop∈store-sops is′. valid-sop sop

assumes prog-step-preserves-valid:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); j⊢p $\rightarrow_{\mathsf{p}}$ (p′,is′); valid ts⟧ ⟹
valid (ts[i:=(p′,is@is′,j,sb@[$\mathrm{Prog}_{\mathsf{sb}}$ p p′ is′],$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])

assumes flush-step-preserves-valid:
⟦i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$); (m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m′,sb′,$\mathcal{O}$′,$\mathcal{R}$′,$\mathcal{S}$′); valid ts⟧ ⟹
valid (ts[i:=(p,is,j,sb′,$\mathcal{D}$,$\mathcal{O}$′,$\mathcal{R}$′)])

assumes sbh-step-preserves-valid:
⟦i < length ts;

ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
(is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is′,j′,sb′,m′,$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$);
valid ts⟧
$\Longrightarrow$
valid (ts[i:=(p,is′,j′,sb′,$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)])

lemma refl′: x = y $\Longrightarrow$ r^** x y
⟨*proof*⟩

lemma no-volatile-Read$_{\mathsf{sb}}$-volatile-reads-consistent:
$\bigwedge$m. outstanding-refs is-volatile-Read$_{\mathsf{sb}}$ sb = {} $\Longrightarrow$ volatile-reads-consistent m sb
⟨*proof*⟩

theorem (**in** program) flush-store-buffer-append:
shows $\bigwedge$ts p m j $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ $\mathcal{S}$ is $\mathcal{O}'$.
⟦i < length ts;
instrs (sb@sb′) @ is$_{\mathsf{sb}}$ = is @ prog-instrs (sb@sb′);
causal-program-history is$_{\mathsf{sb}}$ (sb@sb′);
ts!i = (p,is,j |‘ (dom j − read-tmps (sb@sb′)),x,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
p=hd-prog p$_{\mathsf{sb}}$ (sb@sb′);
(last-prog p$_{\mathsf{sb}}$ (sb@sb′)) = p$_{\mathsf{sb}}$;
reads-consistent True $\mathcal{O}'$ m sb;
history-consistent j p (sb@sb′);
$\forall$sop $\in$ write-sops sb. valid-sop sop;
distinct-read-tmps (sb@sb′);
volatile-reads-consistent m sb
⟧
$\Longrightarrow$
$\exists$is′. instrs sb′ @ is$_{\mathsf{sb}}$ = is′ @ prog-instrs sb′ $\wedge$
(ts,m,$\mathcal{S}$) $\Rightarrow_{\mathsf{d}}^*$
(ts[i:=(last-prog (hd-prog p$_{\mathsf{sb}}$ sb′) sb,is′,j|‘ (dom j − read-tmps sb′),x,
($\mathcal{D}$ $\vee$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb $\neq$ {}),
acquired True sb $\mathcal{O}$, release sb (dom $\mathcal{S}$) $\mathcal{R}$)], flush sb m,share sb $\mathcal{S}$)
⟨*proof*⟩

corollary (**in** program) flush-store-buffer:
assumes i-bound: i < length ts
assumes instrs: instrs sb @ is$_{\mathsf{sb}}$ = is @ prog-instrs sb
assumes cph: causal-program-history is$_{\mathsf{sb}}$ sb
assumes ts-i: ts!i = (p,is,j |‘ (dom j − read-tmps sb),x,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes p: p=hd-prog p$_{\mathsf{sb}}$ sb
assumes last-prog: (last-prog p$_{\mathsf{sb}}$ sb) = p$_{\mathsf{sb}}$
assumes reads-consis: reads-consistent True $\mathcal{O}'$ m sb
assumes hist-consis: history-consistent j p sb
assumes valid-sops: $\forall$sop $\in$ write-sops sb. valid-sop sop

assumes dist: distinct-read-tmps sb
assumes vol-read-consis: volatile-reads-consistent m sb
shows (ts,m,$\mathcal{S}$) $\Rightarrow_{\mathsf{d}}{}^{*}$
(ts[i:=(p_{sb},is_{sb}, j,x,
$\mathcal{D}$ $\vee$ outstanding-refs is-volatile-$\mathrm{Write}_{\mathsf{sb}}$ sb $\neq$ {},acquired True sb $\mathcal{O}$, release sb (dom $\mathcal{S}$) $\mathcal{R}$)],
flush sb m,share sb $\mathcal{S}$)
⟨*proof*⟩

lemma last-prog-same-append: $\bigwedge$xs p_{sb}. last-prog p_{sb} (sb@xs) = p_{sb} $\Longrightarrow$ last-prog p_{sb} xs = p_{sb}
⟨*proof*⟩

lemma reads-consistent-drop-volatile-writes-no-volatile-reads:
$\bigwedge$pending-write $\mathcal{O}$ m. reads-consistent pending-write $\mathcal{O}$ m sb $\Longrightarrow$
outstanding-refs is-volatile-$\mathrm{Read}_{\mathsf{sb}}$ ((dropWhile (Not $\circ$ is-volatile-$\mathrm{Write}_{\mathsf{sb}}$)) sb) = {}
⟨*proof*⟩

lemma reads-consistent-flush-other:
assumes no-volatile-$\mathrm{Write}_{\mathsf{sb}}$-sb: outstanding-refs is-volatile-$\mathrm{Write}_{\mathsf{sb}}$ sb = {}
shows $\bigwedge$m pending-write $\mathcal{O}$.
⟦outstanding-refs (Not $\circ$ is-volatile-$\mathrm{Read}_{\mathsf{sb}}$) xs $\cap$ outstanding-refs is-non-volatile-$\mathrm{Write}_{\mathsf{sb}}$ sb = {};
reads-consistent pending-write $\mathcal{O}$ m xs⟧ $\Longrightarrow$ reads-consistent pending-write $\mathcal{O}$ (flush sb m) xs
⟨*proof*⟩

lemma reads-consistent-flush-independent:
assumes no-volatile-$\mathrm{Write}_{\mathsf{sb}}$-sb: outstanding-refs is-$\mathrm{Write}_{\mathsf{sb}}$ sb $\cap$ outstanding-refs is-non-volatile-$\mathrm{Read}_{\mathsf{sb}}$ xs = {}
assumes consis: reads-consistent pending-write $\mathcal{O}$ m xs
shows reads-consistent pending-write $\mathcal{O}$ (flush sb m) xs
⟨*proof*⟩

lemma reads-consistent-flush-all-until-volatile-write-aux:
assumes no-reads: outstanding-refs is-volatile-$\mathrm{Read}_{\mathsf{sb}}$ xs = {}
shows $\bigwedge$m pending-write $\mathcal{O}'$. ⟦reads-consistent pending-write $\mathcal{O}'$ m xs; $\forall$i < length ts.
let (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) = ts!i in
outstanding-refs (Not $\circ$ is-volatile-$\mathrm{Read}_{\mathsf{sb}}$) xs $\cap$
outstanding-refs is-non-volatile-$\mathrm{Write}_{\mathsf{sb}}$ (takeWhile (Not $\circ$ is-volatile-$\mathrm{Write}_{\mathsf{sb}}$) sb) = {}⟧
$\Longrightarrow$ reads-consistent pending-write $\mathcal{O}'$ (flush-all-until-volatile-write ts m) xs
⟨*proof*⟩

lemma reads-consistent-flush-other′:
assumes no-volatile-Write$_{sb}$-sb: outstanding-refs is-volatile-Write$_{sb}$ sb = {}
shows $\bigwedge$m $\mathcal{O}$.
⟦outstanding-refs is-non-volatile-Write$_{sb}$ sb ∩
(outstanding-refs is-volatile-Write$_{sb}$ xs ∪
outstanding-refs is-non-volatile-Write$_{sb}$ xs ∪
outstanding-refs is-non-volatile-Read$_{sb}$ (dropWhile (Not ∘ is-volatile-Write$_{sb}$) xs) ∪
(outstanding-refs is-non-volatile-Read$_{sb}$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) xs) − RO) ∪
($\mathcal{O}$ ∪ all-acquired (takeWhile (Not ∘ is-volatile-Write$_{sb}$) xs))
) = {};
reads-consistent False $\mathcal{O}$ m xs;
read-only-reads $\mathcal{O}$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) xs) ⊆ RO⟧
⟹ reads-consistent False $\mathcal{O}$ (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 $\mathcal{O}'$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) xs) ⊆ RO
shows $\bigwedge$m. ⟦reads-consistent False $\mathcal{O}'$ m xs; ∀ i < length ts.
let (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$) = ts!i in
outstanding-refs is-non-volatile-Write$_{sb}$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) ∩
(outstanding-refs is-volatile-Write$_{sb}$ xs ∪
outstanding-refs is-non-volatile-Write$_{sb}$ xs ∪
outstanding-refs is-non-volatile-Read$_{sb}$ (dropWhile (Not ∘ is-volatile-Write$_{sb}$) xs) ∪
(outstanding-refs is-non-volatile-Read$_{sb}$ (takeWhile (Not ∘ is-volatile-Write$_{sb}$) xs) − RO) ∪
($\mathcal{O}'$ ∪ all-acquired (takeWhile (Not ∘ is-volatile-Write$_{sb}$) xs))
)
= {}
⟧
⟹ reads-consistent False $\mathcal{O}'$ (flush-all-until-volatile-write ts m) xs
⟨*proof*⟩

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

$\Longrightarrow$ C
⟨*proof*⟩

lemma dropWhile-Cons: (dropWhile P xs) = x#ys $\Longrightarrow$ $\neg$ P x
⟨*proof*⟩

lemma reads-consistent-dropWhile:
reads-consistent pending-write $\mathcal{O}$ m (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) =
reads-consistent True $\mathcal{O}$ m (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
⟨*proof*⟩

theorem
reads-consistent-flush-all-until-volatile-write:
$\bigwedge$i m pending-write. ⟦valid-ownership-and-sharing $\mathcal{S}$ ts;
i < length ts; ts!i = (p, is,j, sb, $\mathcal{D}$, $\mathcal{O}$,$\mathcal{R}$);
reads-consistent pending-write $\mathcal{O}$ m sb ⟧
$\Longrightarrow$ reads-consistent True (acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$)
(flush-all-until-volatile-write ts m) (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
⟨*proof*⟩

lemma split-volatile-Write$_{\mathsf{sb}}$-in-outstanding-refs:
a $\in$ outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ xs $\Longrightarrow$ ($\exists$sop v ys zs A L R W. xs = ys@(Write$_{\mathsf{sb}}$ True a sop v A L R W#zs))
⟨*proof*⟩

lemma sharing-consistent-mono-shared:
$\bigwedge\mathcal{S}\ \mathcal{S}'\ \mathcal{O}$.
dom $\mathcal{S}$ $\subseteq$ dom $\mathcal{S}'$ $\Longrightarrow$ sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ sharing-consistent $\mathcal{S}'$ $\mathcal{O}$ sb
⟨*proof*⟩

lemma sharing-consistent-mono-owns:
$\bigwedge\mathcal{O}\ \mathcal{O}'\ \mathcal{S}$.
$\mathcal{O} \subseteq \mathcal{O}'$ $\Longrightarrow$ sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ sharing-consistent $\mathcal{S}$ $\mathcal{O}'$ sb
⟨*proof*⟩

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

lemma sharing-consistent-all-shared:
$\bigwedge\mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ all-shared sb $\subseteq$ dom $\mathcal{S}$ $\cup$ $\mathcal{O}$

$\langle proof \rangle$

lemma sharing-consistent-share-all-shared:
$\bigwedge \mathcal{S}$. dom (share sb $\mathcal{S}$) $\subseteq$ dom $\mathcal{S}$ $\cup$ all-shared sb
$\langle proof \rangle$

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

lemma all-unshared-append: all-unshared (xs @ ys) = all-unshared xs $\cup$ all-unshared ys
$\langle proof \rangle$

lemma freshly-shared-owned:
$\bigwedge \mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ dom (share sb $\mathcal{S}$) $-$ dom $\mathcal{S}$ $\subseteq$ $\mathcal{O}$
$\langle proof \rangle$

lemma unshared-all-unshared:
$\bigwedge \mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ dom $\mathcal{S}$ $-$ dom (share sb $\mathcal{S}$) $\subseteq$ all-unshared sb
$\langle proof \rangle$

lemma unshared-acquired-or-owned:
$\bigwedge \mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ all-unshared sb $\subseteq$ all-acquired sb $\cup$ $\mathcal{O}$
$\langle proof \rangle$

lemma all-shared-acquired-or-owned:
$\bigwedge \mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ all-shared sb $\subseteq$ all-acquired sb $\cup$ $\mathcal{O}$
$\langle proof \rangle$

lemma sharing-consistent-preservation:
$\bigwedge \mathcal{S}\ \mathcal{S}'\ \mathcal{O}$.
$[\![$sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb;
all-acquired sb $\cap$ dom $\mathcal{S}$ $-$ dom $\mathcal{S}'$ = {};
all-unshared sb $\cap$ dom $\mathcal{S}'$ $-$ dom $\mathcal{S}$ = {}$]\!]$
$\Longrightarrow$ sharing-consistent $\mathcal{S}'$ $\mathcal{O}$ sb
$\langle proof \rangle$

lemma (**in** sharing-consis) sharing-consis-preservation:
assumes dist:

$\forall$ i < length ts. let (-,-,-,sb,-,-,-) = ts!i in
all-acquired sb $\cap$ dom $\mathcal{S}$ − dom $\mathcal{S}'$ = {} $\wedge$ all-unshared sb $\cap$ dom $\mathcal{S}'$ − dom $\mathcal{S}$ =
{}
shows sharing-consis $\mathcal{S}'$ ts
⟨*proof*⟩

lemma (**in** sharing-consis) sharing-consis-shared-exchange:
assumes dist:
$\forall$ i < length ts. let (-,-,-,sb,-,-,-) = ts!i in
$\forall$ a $\in$ all-acquired sb. $\mathcal{S}'$ a = $\mathcal{S}$ a
shows sharing-consis $\mathcal{S}'$ 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 $\mathcal{S}$ $\mathcal{O}$ sb
shows acquired pending-write sb $\mathcal{O}$ $\cup$ dom (share sb $\mathcal{S}$) $\subseteq$ $\mathcal{O}$ $\cup$ dom $\mathcal{S}$
⟨*proof*⟩

lemma acquired-owns-shared:
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows acquired True sb $\mathcal{O}$ $\subseteq$ $\mathcal{O}$ $\cup$ dom $\mathcal{S}$
⟨*proof*⟩

lemma share-owns-shared:
assumes consis: sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb
shows dom (share sb $\mathcal{S}$) $\subseteq$ $\mathcal{O}$ $\cup$ dom $\mathcal{S}$
⟨*proof*⟩

lemma all-shared-append: all-shared (xs@ys) = all-shared xs $\cup$ all-shared ys
⟨*proof*⟩

lemma acquired-union-notin-first:
$\bigwedge$ pending-write A B. a $\in$ acquired pending-write sb (A $\cup$ B) $\Longrightarrow$ a $\notin$ A $\Longrightarrow$ a $\in$ acquired pending-write sb B
⟨*proof*⟩

lemma split-all-acquired-in:
$a \in$ all-acquired xs $\Longrightarrow$
($\exists$sop a$'$ v ys zs A L R W. xs = ys @ Write$_{sb}$ True a$'$ sop v A L R W# zs $\wedge$ a $\in$ A) $\vee$
($\exists$A L R W ys zs. xs = ys @ Ghost$_{sb}$ A L R W# zs $\wedge$ a $\in$ A)
⟨*proof*⟩

lemma split-Write$_{sb}$-in-outstanding-refs:
a $\in$ outstanding-refs is-Write$_{sb}$ xs $\Longrightarrow$ ($\exists$sop volatile v ys zs A L R W. xs = ys@(Write$_{sb}$ volatile a sop v A L R W#zs))
⟨*proof*⟩

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)
⟨*proof*⟩

lemma rtranclp-r-rtranclp: $[\![$r** x y; r y z$]\!]$ $\Longrightarrow$ r** x z
⟨*proof*⟩

lemma r-rtranclp-rtranclp: $[\![$r x y; r** y z$]\!]$ $\Longrightarrow$ r** x z
⟨*proof*⟩

lemma unshared-is-non-volatile-Write$_{sb}$: $\bigwedge\mathcal{S}$.
$[\![$non-volatile-writes-unshared $\mathcal{S}$ sb; a $\in$ dom $\mathcal{S}$; a $\notin$ all-unshared sb$]\!]$ $\Longrightarrow$
a $\notin$ outstanding-refs is-non-volatile-Write$_{sb}$ sb
⟨*proof*⟩

lemma outstanding-non-volatile-Read$_{sb}$-acquired-or-read-only-reads:
$\bigwedge\mathcal{O}$ $\mathcal{S}$ pending-write.
$[\![$non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb;
a $\in$ outstanding-refs is-non-volatile-Read$_{sb}$ sb$]\!]$
$\Longrightarrow$ a $\in$ acquired-reads True sb $\mathcal{O}$ $\vee$ a $\in$ read-only-reads $\mathcal{O}$ sb
⟨*proof*⟩

lemma acquired-reads-union: $\bigwedge$pending-writes A B.
$[\![$a $\in$ acquired-reads pending-writes sb (A $\cup$ B); a $\notin$ A$]\!]$ $\Longrightarrow$ a $\in$ acquired-reads pending-writes sb B
⟨*proof*⟩

lemma non-volatile-writes-unshared-no-outstanding-non-volatile-Write$_{sb}$: $\bigwedge\mathcal{S}$ $\mathcal{S}'$.
$[\![$non-volatile-writes-unshared $\mathcal{S}$ sb;
$\forall$a $\in$ dom $\mathcal{S}'$ $-$ dom $\mathcal{S}$. a $\notin$ outstanding-refs is-non-volatile-Write$_{sb}$ sb $]\!]$
$\Longrightarrow$ non-volatile-writes-unshared $\mathcal{S}'$ sb
⟨*proof*⟩

theorem sharing-consis-share-all-until-volatile-write:
$\bigwedge\mathcal{S}$ ts$'$. ⟦ownership-distinct ts; sharing-consis $\mathcal{S}$ ts; length ts$'$ = length ts;
$\forall$ i < length ts.
(let (-,-,-,sb,-,$\mathcal{O}$,-) = ts!i;
(-,-,-,sb$'$,-,$\mathcal{O}'$,-) = ts$'$!i
in $\mathcal{O}'$ = acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$ $\wedge$
sb$'$ = dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)⟧ $\Longrightarrow$
sharing-consis (share-all-until-volatile-write ts $\mathcal{S}$) ts$'$ $\wedge$
dom (share-all-until-volatile-write ts $\mathcal{S}$) − dom $\mathcal{S}$ $\subseteq$
$\bigcup$ ((λ(-,-,-,-,-,$\mathcal{O}$,-). $\mathcal{O}$) ‘ set ts) $\wedge$
dom $\mathcal{S}$ − dom (share-all-until-volatile-write ts $\mathcal{S}$) $\subseteq$
$\bigcup$ ((λ(-,-,-,sb,-,$\mathcal{O}$,-). all-acquired sb $\cup$ $\mathcal{O}$) ‘ set ts)
⟨*proof*⟩

corollary sharing-consistent-share-all-until-volatile-write:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
shows sharing-consistent (share-all-until-volatile-write ts $\mathcal{S}$)
(acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$)
(dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
⟨*proof*⟩

lemma restrict-map-UNIV [simp]: S |‘ UNIV = S
⟨*proof*⟩

lemma share-all-until-volatile-write-Read-commute:
shows $\bigwedge$S i. ⟦i < length ls; ls!i=(p,Read volatile a t#is,j,sb,$\mathcal{D}$,$\mathcal{O}$)
⟧
$\Longrightarrow$
share-all-until-volatile-write
(ls[i := (p,is, j(t$\mapsto$v), sb @ [Read$_{\mathsf{sb}}$ volatile a t v],$\mathcal{D}'$, $\mathcal{O}$)]) S =
share-all-until-volatile-write ls S
⟨*proof*⟩

lemma share-all-until-volatile-write-Write-commute:
shows $\bigwedge$S i. ⟦i < length ls; ls!i=(p,Write volatile a (D,f) A L R W#is,j,sb,$\mathcal{D}$,$\mathcal{O}$)
⟧
$\Longrightarrow$
share-all-until-volatile-write
(ls[i := (p,is,j, sb @ [Write$_{\mathsf{sb}}$ volatile a t (f j) A L R W], $\mathcal{D}'$, $\mathcal{O}$)]) S =

share-all-until-volatile-write ls S
⟨*proof*⟩

lemma share-all-until-volatile-write-RMW-commute:
shows $\bigwedge$S i. ⟦i < length ls; ls!i=(p,RMW a t (D,f) cond ret A L R W#is,j,[],$\mathcal{D}$,$\mathcal{O}$)
⟧
$\Longrightarrow$
share-all-until-volatile-write (ls[i := (p′,is, j′, [],$\mathcal{D}'$, $\mathcal{O}'$)]) S =
share-all-until-volatile-write ls S
⟨*proof*⟩

lemma share-all-until-volatile-write-Fence-commute:
shows $\bigwedge$S i. ⟦i < length ls; ls!i=(p,Fence#is,j,[],$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
⟧
$\Longrightarrow$
share-all-until-volatile-write (ls[i := (p,is,j, [], $\mathcal{D}'$, $\mathcal{O}$,$\mathcal{R}'$)]) S =
share-all-until-volatile-write ls S
⟨*proof*⟩

lemma unshared-share-in: $\bigwedge$S. a ∈ dom S $\Longrightarrow$ a ∉ all-unshared sb $\Longrightarrow$ a ∈ dom (share sb S)
⟨*proof*⟩

lemma dom-eq-dom-share-eq: $\bigwedge$S S′. dom S = dom S′ $\Longrightarrow$ dom (share sb S) = dom (share sb S′)
⟨*proof*⟩

lemma share-union:
$\bigwedge$A B. ⟦a ∈ dom (share sb (A $\oplus_{\mathsf{z}}$ B)); a ∉ dom A⟧ $\Longrightarrow$ a ∈ dom (share sb (Map.empty $\oplus_{\mathsf{z}}$ B))
⟨*proof*⟩

lemma share-unshared-in:
$\bigwedge$S. a ∈ dom (share sb S) $\Longrightarrow$ a ∈ dom (share sb Map.empty) ∨ (a ∈ dom S ∧ a ∉ all-unshared sb)
⟨*proof*⟩

lemma dom-augment-rels-shared-eq: dom (augment-rels S R $\mathcal{R}$) = dom (augment-rels S′ R $\mathcal{R}$)
⟨*proof*⟩

lemma dom-eq-SomeD1: dom m = dom n $\Longrightarrow$ m x = Some y $\Longrightarrow$ n x $\neq$ None
⟨*proof*⟩

lemma dom-eq-SomeD2: dom m = dom n $\Longrightarrow$ n x = Some y $\Longrightarrow$ m x $\neq$ None
⟨*proof*⟩

lemma dom-augment-rels-rels-eq: dom $\mathcal{R}'$ = dom $\mathcal{R}$ $\Longrightarrow$ dom (augment-rels S R $\mathcal{R}'$) = dom (augment-rels S R $\mathcal{R}$)
⟨*proof*⟩

lemma dom-release-rels-eq: $\bigwedge\mathcal{S}\ \mathcal{R}\ \mathcal{R}'$. dom $\mathcal{R}'$ = dom $\mathcal{R}$ $\Longrightarrow$
dom (release sb $\mathcal{S}$ $\mathcal{R}'$) = dom (release sb $\mathcal{S}$ $\mathcal{R}$)
⟨*proof*⟩

lemma dom-release-shared-eq: $\bigwedge\mathcal{S}\ \mathcal{S}'\ \mathcal{R}$. dom (release sb $\mathcal{S}'$ $\mathcal{R}$) = dom (release sb $\mathcal{S}$ $\mathcal{R}$)
⟨*proof*⟩

lemma share-other-untouched:
$\bigwedge\mathcal{O}\ \mathcal{S}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ a $\notin$ $\mathcal{O}$ $\cup$ all-acquired sb$\Longrightarrow$ share sb $\mathcal{S}$ a = $\mathcal{S}$ a
⟨*proof*⟩

lemma shared-owned: $\bigwedge\mathcal{O}\ \mathcal{S}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ a $\notin$ dom $\mathcal{S}$ $\Longrightarrow$ a $\in$ dom (share sb $\mathcal{S}$) $\Longrightarrow$
a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb
⟨*proof*⟩

lemma share-all-shared-in: a $\in$ dom (share sb $\mathcal{S}$) $\Longrightarrow$ a $\in$ dom $\mathcal{S}$ $\vee$ a $\in$ all-shared sb
⟨*proof*⟩

lemma share-all-until-volatile-write-unowned:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes other: $\forall$ i p is j sb $\mathcal{D}$ $\mathcal{O}$ $\mathcal{R}$. i < length ts $\longrightarrow$ ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
a $\notin$ $\mathcal{O}$ $\cup$ all-acquired sb
shows share-all-until-volatile-write ts $\mathcal{S}$ a = $\mathcal{S}$ a
⟨*proof*⟩

lemma share-shared-eq: $\bigwedge\mathcal{S}'\ \mathcal{S}$. $\mathcal{S}'$ a = $\mathcal{S}$ a $\Longrightarrow$ share sb $\mathcal{S}'$ a = share sb $\mathcal{S}$ a
⟨*proof*⟩

lemma share-all-until-volatile-write-thread-local:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-owned: a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb
shows share-all-until-volatile-write ts $\mathcal{S}$ a = share (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{S}$ a
$\langle proof \rangle$

lemma share-all-until-volatile-write-thread-local$'$:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-owned: a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb
shows share (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) (share-all-until-volatile-write ts $\mathcal{S}$) a =
share sb $\mathcal{S}$ a
$\langle proof \rangle$

lemma (**in** ownership-distinct) in-shared-sb-share-all-until-volatile-write:
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-owned: a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb
assumes a-share: a $\in$ dom (share sb $\mathcal{S}$)
shows a $\in$ dom (share (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
(share-all-until-volatile-write ts $\mathcal{S}$))
$\langle proof \rangle$

lemma owns-unshared-share-acquired:
$\bigwedge\mathcal{S}$ $\mathcal{O}$. $[\![$sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb; a $\in$ $\mathcal{O}$; a $\notin$ all-unshared sb$]\!]$
$\Longrightarrow$ a $\in$ dom (share sb $\mathcal{S}$) $\cup$ acquired True sb $\mathcal{O}$
$\langle proof \rangle$

lemma shared-share-acquired: $\bigwedge\mathcal{S}$ $\mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$
a $\in$ dom $\mathcal{S}$ $\Longrightarrow$ a $\in$ dom (share sb $\mathcal{S}$) $\cup$ acquired True sb $\mathcal{O}$
$\langle proof \rangle$

lemma dom-release-takeWhile:
$\bigwedge$S $\mathcal{R}$.
dom (release (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) S $\mathcal{R}$) =
dom $\mathcal{R}$ $\cup$ all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
$\langle proof \rangle$

lemma share-all-until-volatile-write-share-acquired:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts

assumes a-notin: $a \notin \text{dom } \mathcal{S}$
assumes a-in: $a \in \text{dom (share-all-until-volatile-write ts } \mathcal{S})$
shows $\exists\, i < \text{length ts.}$
let (-,-,-,sb,-,-,-) = ts!i
in $a \in$ all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)
⟨*proof*⟩

lemma all-shared-share-acquired: $\bigwedge\mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}\ \mathcal{O}$ sb $\Longrightarrow$
$a \in$ all-shared sb $\Longrightarrow a \in$ dom (share sb $\mathcal{S}$) $\cup$ acquired True sb $\mathcal{O}$
⟨*proof*⟩

lemma (**in** ownership-distinct) share-all-until-volatile-write-share-acquired:
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes i-bound: $i <$ length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-in: $a \in$ dom (share-all-until-volatile-write ts $\mathcal{S}$)
shows $a \in$ dom (share sb $\mathcal{S}$) $\lor$ $a \in$ acquired True sb $\mathcal{O}$ $\lor$
($\exists\, j <$ length ts. $j \neq i \land$
(let (-,-,-,sb$_j$,-,-,-) = ts!j
in $a \in$ all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$)))
⟨*proof*⟩

lemma acquired-all-shared-in:
$\bigwedge$A. $a \in$ acquired True sb A $\Longrightarrow a \in$ acquired True sb {} $\lor$ ($a \in A \land a \notin$ all-shared sb)
⟨*proof*⟩

lemma all-shared-acquired-in: $\bigwedge$A. $a \in A \Longrightarrow a \notin$ all-shared sb $\Longrightarrow a \in$ acquired True sb A
⟨*proof*⟩

lemma owned-share-acquired: $\bigwedge\mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}\ \mathcal{O}$ sb $\Longrightarrow$
$a \in \mathcal{O} \Longrightarrow a \in$ dom (share sb $\mathcal{S}$) $\cup$ acquired True sb $\mathcal{O}$
⟨*proof*⟩

lemma outstanding-refs-non-volatile-Read$_{sb}$-all-acquired:
$\bigwedge$m $\mathcal{S}\ \mathcal{O}$ pending-write.
⟦reads-consistent pending-write $\mathcal{O}$ m sb;non-volatile-owned-or-read-only pending-write $\mathcal{S}\ \mathcal{O}$ sb;
$a \in$ outstanding-refs is-non-volatile-Read$_{sb}$ sb⟧
$\Longrightarrow a \in\ \mathcal{O} \lor a \in$ all-acquired sb $\lor$
$a \in$ read-only-reads $\mathcal{O}$ sb
⟨*proof*⟩

lemma outstanding-refs-non-volatile-Read$_{sb}$-all-acquired-dropWhile:
assumes consis: reads-consistent pending-write $\mathcal{O}$ m sb
assumes nvo: non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb
assumes out: a $\in$ outstanding-refs is-non-volatile-Read$_{sb}$ (dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)
shows a $\in \mathcal{O} \vee$ a $\in$ all-acquired sb $\vee$
a $\in$ read-only-reads $\mathcal{O}$ sb
$\langle proof \rangle$

lemma share-commute:
$\bigwedge$L R $\mathcal{S}$ $\mathcal{O}$. $[\![$sharing-consistent $\mathcal{S}$ $\mathcal{O}$ 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 = {}$]\!] \Longrightarrow$
(share sb ($\mathcal{S} \oplus_{W}$ R $\ominus_{A}$ L)) =
(share sb $\mathcal{S}$) $\oplus_{W}$ R $\ominus_{A}$ L
$\langle proof \rangle$

lemma share-all-until-volatile-write-commute:
$\bigwedge$ $\mathcal{S}$ R L. $[\![$ownership-distinct ts; sharing-consis $\mathcal{S}$ ts;
$\forall$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. i < length ts $\longrightarrow$ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ L = {};
$\forall$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. i < length ts $\longrightarrow$ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ A = {};
$\forall$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. i < length ts $\longrightarrow$ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-acquired (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ R = {};
$\forall$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. i < length ts $\longrightarrow$ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-unshared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ R = {};
$\forall$i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. i < length ts $\longrightarrow$ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\cap$ R = {}$]\!]$
$\Longrightarrow$
share-all-until-volatile-write ts $\mathcal{S} \oplus_{W}$ R $\ominus_{A}$ L = share-all-until-volatile-write ts ($\mathcal{S} \oplus_{W}$ R $\ominus_{A}$ L)
$\langle proof \rangle$

lemma share-append-Ghost$_{sb}$:
$\bigwedge\mathcal{S}$. outstanding-refs is-volatile-Write$_{sb}$ sb = {} $\Longrightarrow$ (share (sb @ [Ghost$_{sb}$ A L R W]) $\mathcal{S}$) = (share sb $\mathcal{S}$) $\oplus_{W}$ R $\ominus_{A}$ L
$\langle proof \rangle$

lemma share-append-Ghost$_{sb}$':
$\bigwedge\mathcal{S}$. outstanding-refs is-volatile-Write$_{sb}$ sb $\neq$ {} $\Longrightarrow$
(share (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) (sb @ [Ghost$_{sb}$ A L R W])) $\mathcal{S}$) =
(share (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\mathcal{S}$)
$\langle proof \rangle$

lemma share-all-until-volatile-write-append-Ghost$_{sb}$:

assumes no-out-VWrite$_{\mathsf{sb}}$: outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb = {}
shows $\bigwedge\mathcal{S}$ i. ⟦ownership-distinct ts; sharing-consis $\mathcal{S}$ ts;
i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
$\forall$j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. j < length ts $\longrightarrow$ i$\neq$j $\longrightarrow$ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\cap$ L = {};
$\forall$j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. j < length ts $\longrightarrow$ i$\neq$j $\longrightarrow$ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\cap$ A = {};
$\forall$j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. j < length ts $\longrightarrow$ i$\neq$j $\longrightarrow$ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-acquired (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\cap$ R = {};
$\forall$j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. j < length ts $\longrightarrow$ i$\neq$j $\longrightarrow$ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-unshared (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\cap$ R = {};
$\forall$j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. j < length ts $\longrightarrow$ i$\neq$j $\longrightarrow$ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $\longrightarrow$
all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\cap$ R = {}⟧
$\Longrightarrow$
share-all-until-volatile-write (ts[i := (p′, is′,j′, sb @ [Ghost$_{\mathsf{sb}}$ A L R W], $\mathcal{D}'$, $\mathcal{O}'$)]) $\mathcal{S}$
= share-all-until-volatile-write ts $\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L
⟨*proof*⟩

lemma share-domain-changes:
$\bigwedge\mathcal{S}$ $\mathcal{S}'$. a $\in$ all-shared sb $\cup$ all-unshared sb $\Longrightarrow$ share sb $\mathcal{S}'$ a = share sb $\mathcal{S}$ a
⟨*proof*⟩

lemma share-domain-changesX:
$\bigwedge\mathcal{S}$ $\mathcal{S}'$ X. $\forall$a $\in$ X. $\mathcal{S}'$ a = $\mathcal{S}$ a
$\Longrightarrow$ a $\in$ all-shared sb $\cup$ all-unshared sb $\cup$ X $\Longrightarrow$ share sb $\mathcal{S}'$ a = share sb $\mathcal{S}$ a
⟨*proof*⟩

lemma share-unchanged:
$\bigwedge\mathcal{S}$. a $\notin$ all-shared sb $\cup$ all-unshared sb $\cup$ all-acquired sb $\Longrightarrow$ share sb $\mathcal{S}$ a = $\mathcal{S}$ a
⟨*proof*⟩

lemma share-augment-release-commute:
assumes dist: (R $\cup$ L $\cup$ A) $\cap$ (all-shared sb $\cup$ all-unshared sb $\cup$ all-acquired sb) = {}
shows (share sb $\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) = share sb ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)
⟨*proof*⟩

lemma share-append-commute:
$\bigwedge$ys $\mathcal{S}$. (all-shared xs $\cup$ all-unshared xs $\cup$ all-acquired xs) $\cap$
(all-shared ys $\cup$ all-unshared ys $\cup$ all-acquired ys) = {}
$\Longrightarrow$ share xs (share ys $\mathcal{S}$) = share ys (share xs $\mathcal{S}$)
⟨*proof*⟩

lemma share-append-commute′:
assumes dist: (all-shared xs $\cup$ all-unshared xs $\cup$ all-acquired xs) $\cap$

(all-shared ys ∪ all-unshared ys ∪ all-acquired ys) = {}
shows share (ys@xs) $\mathcal{S}$ = share (xs@ys) $\mathcal{S}$
⟨*proof*⟩

lemma share-all-until-volatile-write-share-commute:
shows $\bigwedge$ $\mathcal{S}$ (sb′::′a memref list). ⟦ownership-distinct ts; sharing-consis $\mathcal{S}$ ts;
∀ i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j (sb::′a memref list). i < length ts
⟶ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟶
(all-shared (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) ∪
all-unshared (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) ∪
all-acquired (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb)) ∩
(all-shared sb′ ∪ all-unshared sb′ ∪ all-acquired sb′) = {}⟧
⟹
share-all-until-volatile-write ts (share sb′ $\mathcal{S}$) =
share sb′ (share-all-until-volatile-write ts $\mathcal{S}$)
⟨*proof*⟩

lemma all-shared-takeWhile-subset: all-shared (takeWhile P sb) ⊆ all-shared sb
⟨*proof*⟩
lemma all-shared-dropWhile-subset: all-shared (dropWhile P sb) ⊆ all-shared sb
⟨*proof*⟩

lemma all-unshared-takeWhile-subset: all-unshared (takeWhile P sb) ⊆ all-unshared sb
⟨*proof*⟩
lemma all-unshared-dropWhile-subset: all-unshared (dropWhile P sb) ⊆ all-unshared sb
⟨*proof*⟩

lemma all-acquired-takeWhile-subset: all-acquired (takeWhile P sb) ⊆ all-acquired sb
⟨*proof*⟩
lemma all-acquired-dropWhile-subset: all-acquired (dropWhile P sb) ⊆ all-acquired sb
⟨*proof*⟩

lemma share-all-until-volatile-write-flush-commute:
assumes takeWhile-empty: (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) = []
shows $\bigwedge$ $\mathcal{S}$ R L W A i. ⟦ownership-distinct ts; sharing-consis $\mathcal{S}$ ts; i < length ts;
ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
∀ i p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j (sb::′a memref list). i < length ts
⟶ ts!i=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟶
(all-shared (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) ∪
all-unshared (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) ∪
all-acquired (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb)) ∩
(all-shared (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb′) ∪
all-unshared (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb′) ∪
all-acquired (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb′)) = {};
∀ j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j (sb::′a memref list). j < length ts ⟶ i≠j
⟶ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟶
(all-shared sb ∪ all-unshared sb ∪ all-acquired sb) ∩
(R ∪ L ∪ A) = {}⟧

$\Longrightarrow$
share-all-until-volatile-write (ts[i :=(p′,is′,j′,sb′,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′)]) ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) =
share (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb′) (share-all-until-volatile-write ts $\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)
⟨*proof*⟩

lemma share-all-until-volatile-write-Ghost$_{\mathsf{sb}}$-commute:
shows ⋀ $\mathcal{S}$ i. ⟦ownership-distinct ts; sharing-consis $\mathcal{S}$ ts; i < length ts;
ts!i = (p,is,j,Ghost$_{\mathsf{sb}}$ A L R W#sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$);
∀ j p is $\mathcal{O}$ $\mathcal{R}$ $\mathcal{D}$ j sb. j < length ts ⟶ i≠j ⟶ ts!j=(p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) ⟶
(all-shared (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) ∪ all-unshared (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) ∪ all-acquired (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb)) ∩

(R ∪ L ∪ A) = {}⟧
$\Longrightarrow$
share-all-until-volatile-write (ts[i :=(p′,is′,j′,sb,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′)]) ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L) =
share-all-until-volatile-write ts $\mathcal{S}$
⟨*proof*⟩

lemma share-all-until-volatile-write-update-sb:
assumes congr: ⋀S. share (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb′) S = share (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) S
shows ⋀$\mathcal{S}$ i. ⟦i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟧
$\Longrightarrow$
share-all-until-volatile-write ts $\mathcal{S}$ =
share-all-until-volatile-write (ts[i := (p′, is′,j′, sb′, $\mathcal{D}$′, $\mathcal{O}$′,$\mathcal{R}$′)]) $\mathcal{S}$
⟨*proof*⟩

lemma share-all-until-volatile-write-append-Ghost$_{\mathsf{sb}}$′:
assumes out-VWrite$_{\mathsf{sb}}$: outstanding-refs is-volatile-Write$_{\mathsf{sb}}$ sb ≠ {}
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
shows share-all-until-volatile-write ts $\mathcal{S}$ =
share-all-until-volatile-write
(ts[i := (p′, is′,j′, sb @ [Ghost$_{\mathsf{sb}}$ A L R W], $\mathcal{D}$′, $\mathcal{O}$′,$\mathcal{R}$′)]) $\mathcal{S}$
⟨*proof*⟩

lemma acquired-append-Prog$_{\mathsf{sb}}$:
⋀S. (acquired pending-write (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) (sb @ [Prog$_{\mathsf{sb}}$ p_1 p_2 mis])) S) =
(acquired pending-write (takeWhile (Not ∘ is-volatile-Write$_{\mathsf{sb}}$) sb) S)
⟨*proof*⟩

lemma outstanding-refs-non-empty-dropWhile:
outstanding-refs P xs ≠ {} $\Longrightarrow$ outstanding-refs P (dropWhile (Not ∘ P) xs) ≠ {}
⟨*proof*⟩

lemma ex-not: Ex Not

⟨proof⟩

lemma (**in** computation) concurrent-step-append:
assumes step: $(ts,m,\mathcal{S}) \Rightarrow (ts',m',\mathcal{S}')$
shows $(xs@ts,m,\mathcal{S}) \Rightarrow (xs@ts',m',\mathcal{S}')$
⟨proof⟩

primrec weak-sharing-consistent:: owns ⇒ 'a memref list ⇒ bool
where
weak-sharing-consistent $\mathcal{O}$ [] = True
| weak-sharing-consistent $\mathcal{O}$ (r#rs) =
(case r of
$\text{Write}_{\mathsf{sb}}$ volatile - - - A L R W ⇒
(if volatile then $L \subseteq A \wedge A \cap R = \{\} \wedge R \subseteq \mathcal{O} \wedge$
weak-sharing-consistent $(\mathcal{O} \cup A - R)$ rs
else weak-sharing-consistent $\mathcal{O}$ rs)
| $\text{Ghost}_{\mathsf{sb}}$ A L R W ⇒ $L \subseteq A \wedge A \cap R = \{\} \wedge R \subseteq \mathcal{O} \wedge$ weak-sharing-consistent $(\mathcal{O} \cup A - R)$ rs
| - ⇒ weak-sharing-consistent $\mathcal{O}$ rs)

lemma sharing-consistent-weak-sharing-consistent:
$\bigwedge\mathcal{S}\ \mathcal{O}$. sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb $\Longrightarrow$ weak-sharing-consistent $\mathcal{O}$ sb
⟨proof⟩

lemma weak-sharing-consistent-append:
$\bigwedge\mathcal{O}$. weak-sharing-consistent $\mathcal{O}$ (xs @ ys) =
(weak-sharing-consistent $\mathcal{O}$ xs ∧ weak-sharing-consistent (acquired True xs $\mathcal{O}$) ys)
⟨proof⟩

lemma read-only-share-unowned: $\bigwedge\mathcal{O}\ \mathcal{S}$.
⟦weak-sharing-consistent $\mathcal{O}$ sb; $a \notin \mathcal{O} \cup$ all-acquired sb; $a \in$ read-only (share sb $\mathcal{S}$)⟧
$\Longrightarrow a \in$ read-only $\mathcal{S}$
⟨proof⟩

lemma share-read-only-mono-in:
assumes a-in: $a \in$ read-only (share sb $\mathcal{S}$)
assumes ss: read-only $\mathcal{S} \subseteq$ read-only $\mathcal{S}'$
shows $a \in$ read-only (share sb $\mathcal{S}'$)
⟨*proof*⟩

lemma read-only-unacquired-share:
$\bigwedge S\ \mathcal{O}$. $[\![\mathcal{O} \cap$ read-only $S = \{\}$; weak-sharing-consistent $\mathcal{O}$ sb; $a \in$ read-only S;
$a \notin$ all-acquired sb $]\!]$
$\Longrightarrow a \in$ read-only (share sb S)
⟨*proof*⟩

lemma read-only-share-unacquired: $\bigwedge\ \mathcal{O}$ S. $\mathcal{O} \cap$ read-only S $= \{\} \Longrightarrow$ weak-sharing-consistent $\mathcal{O}$ sb $\Longrightarrow$
$a \in$ read-only (share sb S) $\Longrightarrow a \notin$ acquired True sb $\mathcal{O}$
⟨*proof*⟩

lemma read-only-share-all-acquired-in:
$\bigwedge S\ \mathcal{O}$. $[\![\mathcal{O} \cap$ read-only $S = \{\}$; weak-sharing-consistent $\mathcal{O}$ sb; $a \in$ read-only (share sb S)$]\!]$

$\Longrightarrow a \in$ read-only (share sb Map.empty) $\vee$ ($a \in$ read-only S $\wedge$ $a \notin$ all-acquired sb)
⟨*proof*⟩

lemma weak-sharing-consistent-preserves-distinct:
$\bigwedge\mathcal{O}\ \mathcal{S}$. weak-sharing-consistent $\mathcal{O}$ sb $\Longrightarrow \mathcal{O} \cap$ read-only $\mathcal{S} = \{\} \Longrightarrow$
acquired True sb $\mathcal{O} \cap$ read-only (share sb $\mathcal{S}$) $= \{\}$
⟨*proof*⟩

locale weak-sharing-consis =
fixes ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes weak-sharing-consis:
$\bigwedge$i p is $\mathcal{O}\ \mathcal{R}\ \mathcal{D}$ j sb.
$[\![$i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$) $]\!]$
$\Longrightarrow$
weak-sharing-consistent $\mathcal{O}$ sb

sublocale sharing-consis $\subseteq$ weak-sharing-consis
⟨*proof*⟩

lemma weak-sharing-consis-tl: weak-sharing-consis (t#ts) $\Longrightarrow$ weak-sharing-consis ts
⟨*proof*⟩

lemma read-only-share-all-until-volatile-write-unacquired:
$\bigwedge\mathcal{S}$. [[ownership-distinct ts; read-only-unowned $\mathcal{S}$ ts; weak-sharing-consis ts;
$\forall$ i < length ts. (let (-,-,-,sb,-,$\mathcal{O}$,$\mathcal{R}$) = ts!i in
a $\notin$ all-acquired (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb));
a $\in$ read-only $\mathcal{S}$]]
$\Longrightarrow$ a $\in$ read-only (share-all-until-volatile-write ts $\mathcal{S}$)
⟨*proof*⟩

lemma read-only-share-unowned-in:
[[weak-sharing-consistent $\mathcal{O}$ sb; a $\in$ read-only (share sb $\mathcal{S}$)]]
$\Longrightarrow$ a $\in$ read-only $\mathcal{S}$ $\cup$ $\mathcal{O}$ $\cup$ all-acquired sb
⟨*proof*⟩

lemma read-only-shared-all-until-volatile-write-subset:
$\bigwedge\mathcal{S}$. [[ownership-distinct ts;
weak-sharing-consis ts]] $\Longrightarrow$
read-only (share-all-until-volatile-write ts $\mathcal{S}$) $\subseteq$
read-only $\mathcal{S}$ $\cup$ ($\bigcup$((λ(-, -, -, sb, -, $\mathcal{O}$,-). $\mathcal{O}$ $\cup$ all-acquired (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)) ' set ts))
⟨*proof*⟩

lemma weak-sharing-consistent-preserves-distinct-share-all-until-volatile-write:
$\bigwedge\mathcal{S}$ i. [[ownership-distinct ts; read-only-unowned $\mathcal{S}$ ts;weak-sharing-consis ts;
i < length ts; ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)]]
$\Longrightarrow$ acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$ $\cap$
read-only (share-all-until-volatile-write ts $\mathcal{S}$) = {}
⟨*proof*⟩

lemma in-read-only-share-all-until-volatile-write:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes ro-unowned: read-only-unowned $\mathcal{S}$ ts
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-unacquired-others: $\forall$ j < length ts. i$\neq$j $\longrightarrow$
(let (-,-,-,sb$_{\mathsf{j}}$,-,-,-) = ts!j in
a $\notin$ all-acquired (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb$_{\mathsf{j}}$))
assumes a-ro-share: a $\in$ read-only (share sb $\mathcal{S}$)
shows a $\in$ read-only (share (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
(share-all-until-volatile-write ts $\mathcal{S}$))
⟨*proof*⟩

lemma all-acquired-dropWhile-in: x $\in$ all-acquired (dropWhile P sb) $\Longrightarrow$ x $\in$ all-acquired sb
⟨*proof*⟩

lemma all-acquired-takeWhile-in: x $\in$ all-acquired (takeWhile P sb) $\Longrightarrow$ x $\in$ all-acquired sb
⟨*proof*⟩

lemmas all-acquired-takeWhile-dropWhile-in = all-acquired-takeWhile-in all-acquired-dropWhile-in

lemma split-in-read-only-reads:
$\bigwedge\mathcal{O}$. a $\in$ read-only-reads $\mathcal{O}$ xs $\Longrightarrow$
($\exists$t v ys zs. xs=ys @ Read$_{sb}$ False a t v # zs $\wedge$ a $\notin$ acquired True ys $\mathcal{O}$)
⟨*proof*⟩

lemma insert-monoD: W $\subseteq$ W$'$ $\Longrightarrow$ insert a W $\subseteq$ insert a W$'$
⟨*proof*⟩

primrec unforwarded-non-volatile-reads:: $'$a memref list $\Rightarrow$ addr set $\Rightarrow$ 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 $\wedge$ $\neg$ volatile
then insert a (unforwarded-non-volatile-reads xs W)
else (unforwarded-non-volatile-reads xs W))
| Write$_{sb}$ - a - - - - - - $\Rightarrow$ unforwarded-non-volatile-reads xs (insert a W)
| - $\Rightarrow$ unforwarded-non-volatile-reads xs W)

lemma unforwarded-non-volatile-reads-non-volatile-Read$_{sb}$:
$\bigwedge$W. unforwarded-non-volatile-reads sb W $\subseteq$ outstanding-refs is-non-volatile-Read$_{sb}$ sb
⟨*proof*⟩

lemma in-unforwarded-non-volatile-reads-non-volatile-Read$_{sb}$:
a $\in$ unforwarded-non-volatile-reads sb W $\Longrightarrow$ a $\in$ outstanding-refs is-non-volatile-Read$_{sb}$ sb
⟨*proof*⟩

lemma unforwarded-non-volatile-reads-antimono:
$\bigwedge$W W$'$. W $\subseteq$ W$'$ $\Longrightarrow$ unforwarded-non-volatile-reads xs W$'$ $\subseteq$ unforwarded-non-volatile-reads xs W
⟨*proof*⟩

lemma unforwarded-non-volatile-reads-antimono-in:
x $\in$ unforwarded-non-volatile-reads xs W$'$ $\Longrightarrow$ W $\subseteq$ W$'$
$\Longrightarrow$ x $\in$ 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 ∪
unforwarded-non-volatile-reads ys (W ∪ outstanding-refs is-Write$_{sb}$ xs))
⟨*proof*⟩

lemma reads-consistent-mem-eq-on-unforwarded-non-volatile-reads:
assumes mem-eq: ∀a ∈ A ∪ W. m′ a = m a
assumes subset: unforwarded-non-volatile-reads sb W ⊆ A
assumes consis-m: reads-consistent pending-write $\mathcal{O}$ m sb
shows reads-consistent pending-write $\mathcal{O}$ m′ sb
⟨*proof*⟩

lemma reads-consistent-mem-eq-on-unforwarded-non-volatile-reads-drop:
assumes mem-eq: ∀a ∈ A ∪ W. m′ a = m a
assumes subset: unforwarded-non-volatile-reads (dropWhile (Not ∘ is-volatile-Write$_{sb}$) sb) W ⊆ A
assumes subset-acq: acquired-reads True (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) $\mathcal{O}$ ⊆ A
assumes consis-m: reads-consistent False $\mathcal{O}$ m sb
shows reads-consistent False $\mathcal{O}$ m′ sb
⟨*proof*⟩

lemma read-only-read-witness:$\bigwedge\mathcal{S}$ $\mathcal{O}$.
⟦non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb;
a ∈ read-only-reads $\mathcal{O}$ sb⟧
⟹
∃xs ys t v. sb=xs@ Read$_{sb}$ False a t v # ys ∧ a ∈ read-only (share xs $\mathcal{S}$) ∧ a ∉ read-only-reads $\mathcal{O}$ xs
⟨*proof*⟩

lemma read-only-read-acquired-witness: $\bigwedge\mathcal{S}$ $\mathcal{O}$.
⟦non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb; sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb;
a ∉ read-only $\mathcal{S}$; a ∉ $\mathcal{O}$; a ∈ read-only-reads $\mathcal{O}$ sb⟧
⟹
∃xs ys t v. sb=xs@ Read$_{sb}$ False a t v # ys ∧ a ∈ all-acquired xs ∧ a ∈ read-only (share xs $\mathcal{S}$) ∧
a ∉ read-only-reads $\mathcal{O}$ xs
⟨*proof*⟩

lemma unforwarded-not-written: $\bigwedge$W. a $\in$ unforwarded-non-volatile-reads sb W $\Longrightarrow$ a $\notin$ W
$\langle proof \rangle$

lemma unforwarded-witness:$\bigwedge$X.
 $[\![$a $\in$ unforwarded-non-volatile-reads sb X$]\!]$
 $\Longrightarrow$
 $\exists$xs ys t v. sb=xs@ $\text{Read}_{\mathsf{sb}}$ False a t v # ys $\wedge$ a $\notin$ outstanding-refs is-$\text{Write}_{\mathsf{sb}}$ xs
$\langle proof \rangle$

lemma read-only-read-acquired-unforwarded-witness: $\bigwedge\mathcal{S}$ $\mathcal{O}$ X.
 $[\![$non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb; sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb;
 a $\notin$ read-only $\mathcal{S}$; a $\notin$ $\mathcal{O}$; a $\in$ read-only-reads $\mathcal{O}$ sb;
 a $\in$ unforwarded-non-volatile-reads sb X $]\!]$
 $\Longrightarrow$
 $\exists$xs ys t v. sb=xs@ $\text{Read}_{\mathsf{sb}}$ False a t v # ys $\wedge$ a $\in$ all-acquired xs $\wedge$
   a $\notin$ outstanding-refs is-$\text{Write}_{\mathsf{sb}}$ xs
$\langle proof \rangle$

lemma takeWhile-prefix: $\exists$ys. takeWhile P xs @ ys = xs
$\langle proof \rangle$

lemma unforwarded-empty-extend:
 $\bigwedge$W. x $\in$ unforwarded-non-volatile-reads sb {} $\Longrightarrow$ x $\notin$ W $\Longrightarrow$ x $\in$ unforwarded-non-volatile-reads sb W
$\langle proof \rangle$

lemma notin-unforwarded-empty:
 $\bigwedge$W. a $\notin$ unforwarded-non-volatile-reads sb W $\Longrightarrow$ a $\notin$ W $\Longrightarrow$ a $\notin$ unforwarded-non-volatile-reads sb {}
$\langle proof \rangle$

lemma
 assumes ro: a $\in$ read-only $\mathcal{S}$ $\longrightarrow$ a $\in$ read-only $\mathcal{S}'$
 assumes a-in: a $\in$ read-only ($\mathcal{S}$ $\oplus_{\mathsf{W}}$ R)
 shows a $\in$ read-only ($\mathcal{S}'$ $\oplus_{\mathsf{W}}$ R)
 $\langle proof \rangle$

lemma
 assumes ro: a $\in$ read-only $\mathcal{S}$ $\longrightarrow$ a $\in$ read-only $\mathcal{S}'$
 assumes a-in: a $\in$ read-only ($\mathcal{S}$ $\ominus_{\mathsf{A}}$ L)
 shows a $\in$ read-only ($\mathcal{S}'$ $\ominus_{\mathsf{A}}$ L)
 $\langle proof \rangle$

lemma non-volatile-owned-or-read-only-read-only-reads-eq:
 $\bigwedge\mathcal{S}$ $\mathcal{S}'$ $\mathcal{O}$ pending-write.

$[\![$non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb;
$\forall$ a $\in$ read-only-reads $\mathcal{O}$ sb. a $\in$ read-only $\mathcal{S}$ $\longrightarrow$ a $\in$ read-only $\mathcal{S}'$
$]\!]$
$\Longrightarrow$ non-volatile-owned-or-read-only pending-write $\mathcal{S}'$ $\mathcal{O}$ sb
$\langle proof \rangle$

lemma non-volatile-owned-or-read-only-read-only-reads-eq$'$:
$\bigwedge\mathcal{S}$ $\mathcal{S}'$ $\mathcal{O}$.
$[\![$non-volatile-owned-or-read-only False $\mathcal{S}$ $\mathcal{O}$ sb;
$\forall$ a $\in$ read-only-reads (acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$)
(dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb). a $\in$ read-only $\mathcal{S}$ $\longrightarrow$ a $\in$ read-only $\mathcal{S}'$
$]\!]$
$\Longrightarrow$ non-volatile-owned-or-read-only False $\mathcal{S}'$ $\mathcal{O}$ sb
$\langle proof \rangle$

lemma no-write-to-read-only-memory-read-only-reads-eq:
$\bigwedge\mathcal{S}$ $\mathcal{S}'$.
$[\![$no-write-to-read-only-memory $\mathcal{S}$ sb;
$\forall$ a $\in$ outstanding-refs is-Write$_{\mathsf{sb}}$ sb. a $\in$ read-only $\mathcal{S}'$ $\longrightarrow$ a $\in$ read-only $\mathcal{S}$
$]\!]$
$\Longrightarrow$ no-write-to-read-only-memory $\mathcal{S}'$ sb
$\langle proof \rangle$

lemma reads-consistent-drop:
reads-consistent False $\mathcal{O}$ m sb
$\Longrightarrow$ reads-consistent True
(acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$)
(flush (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) m)
(dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
$\langle proof \rangle$

lemma outstanding-refs-non-volatile-Read$_{\mathsf{sb}}$-all-acquired-dropWhile$'$:
$\bigwedge$m $\mathcal{S}$ $\mathcal{O}$ pending-write.
$[\![$reads-consistent pending-write $\mathcal{O}$ m sb;non-volatile-owned-or-read-only pending-write $\mathcal{S}$ $\mathcal{O}$ sb;
a $\in$ outstanding-refs is-non-volatile-Read$_{\mathsf{sb}}$ (dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)$]\!]$
$\Longrightarrow$ a $\in$ $\mathcal{O}$ $\vee$ a $\in$ all-acquired sb $\vee$
a $\in$ read-only-reads (acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb) $\mathcal{O}$)
(dropWhile (Not $\circ$ is-volatile-Write$_{\mathsf{sb}}$) sb)
$\langle proof \rangle$

end

theory ReduceStoreBufferSimulation

imports ReduceStoreBuffer
begin

locale $\text{initial}_{\text{sb}}$ = 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!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!] \Longrightarrow$ sb=[]
assumes empty-is: $[\![$i < length ts; ts!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!] \Longrightarrow$ is=[]
assumes empty-rels: $[\![$i < length ts; ts!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!] \Longrightarrow$ $\mathcal{R}$=Map.empty

sublocale $\text{initial}_{\text{sb}} \subseteq$ outstanding-non-volatile-refs-owned-or-read-only
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ outstanding-volatile-writes-unowned-by-others
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ read-only-reads-unowned
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ ownership-distinct
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-ownership $\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ outstanding-non-volatile-writes-unshared
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ sharing-consis
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ no-outstanding-write-to-read-only-memory
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-sharing $\langle proof \rangle$
sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-ownership-and-sharing $\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ load-tmps-distinct
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ read-tmps-distinct
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ load-tmps-read-tmps-distinct
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ load-tmps-read-tmps-distinct $\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-write-sops
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-store-sops
$\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-sops $\langle proof \rangle$

sublocale $\text{initial}_{\text{sb}} \subseteq$ valid-reads
$\langle proof \rangle$

sublocale $\text{initial}_{sb} \subseteq$ valid-history
⟨*proof*⟩

sublocale $\text{initial}_{sb} \subseteq$ valid-data-dependency
⟨*proof*⟩

sublocale $\text{initial}_{sb} \subseteq$ load-tmps-fresh
⟨*proof*⟩

sublocale $\text{initial}_{sb} \subseteq$ enough-flushs
⟨*proof*⟩

sublocale $\text{initial}_{sb} \subseteq$ 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};
$\mathcal{S}$ = share-all-until-volatile-write ts_{sb} $\mathcal{S}_{sb}$;
length ts_{sb} = length ts;
∀i < length ts_{sb}.
let (p, is_{sb}, j, sb, $\mathcal{D}_{sb}$, $\mathcal{O}$, $\mathcal{R}$) = ts_{sb}!i;
suspends = dropWhile (Not ∘ is-volatile-Write_{sb}) sb
in ∃is $\mathcal{D}$. instrs suspends @ is_{sb} = is @ prog-instrs suspends ∧
$\mathcal{D}_{sb}$ = ($\mathcal{D}$ ∨ outstanding-refs is-volatile-Write_{sb} sb ≠ {}) ∧
ts!i = (hd-prog p suspends,
is,
j |` (dom j − read-tmps suspends),(),
$\mathcal{D}$,
acquired True (takeWhile (Not ∘ is-volatile-Write_{sb}) sb) $\mathcal{O}$,
release (takeWhile (Not ∘ is-volatile-Write_{sb}) sb) (dom $\mathcal{S}_{sb}$) $\mathcal{R}$)
⟧
⟹
(ts_{sb},m_{sb},$\mathcal{S}_{sb}$) ∼ (ts,m,$\mathcal{S}$)

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
(‹- $\sim_h$ - › [60,60] 100)
where
⟦length ts = length ts_h;
∀i < length ts.
(∃$\mathcal{O}'$ $\mathcal{D}'$ $\mathcal{R}'$.
let (p,is, j, sb,$\mathcal{D}$, $\mathcal{O}$,$\mathcal{R}$) = ts_h!i in
ts!i=(p,is, j, filter is-Write_{sb} sb,$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$) ∧
(filter is-Write_{sb} sb = [] ⟶ sb=[]))
⟧
⟹
ts $\sim_h$ ts_h

lemma (**in** initial_{sb}) history-refl:ts $\sim_h$ ts
⟨*proof*⟩

lemma share-all-empty: ∀i p is xs sb $\mathcal{D}$ $\mathcal{O}$ $\mathcal{R}$. i < length ts ⟶ ts!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)⟶ sb=[]
⟹ share-all-until-volatile-write ts $\mathcal{S}$ = $\mathcal{S}$
⟨*proof*⟩

lemma flush-all-empty: $\forall$ i p is xs sb $\mathcal{D}$ $\mathcal{O}$ $\mathcal{R}$. i < length ts $\longrightarrow$ ts!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$\longrightarrow$ sb=[]
  $\Longrightarrow$ flush-all-until-volatile-write ts m = m
⟨*proof*⟩

lemma sim-config-emptyE:
  assumes empty:
  $\forall$ i p is xs sb $\mathcal{D}$ $\mathcal{O}$ $\mathcal{R}$. i < length ts_{sb} $\longrightarrow$ ts_{sb}!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$\longrightarrow$ sb=[]
  assumes sim: (ts_{sb},m_{sb},$\mathcal{S}_{sb}$) $\sim$ (ts,m,$\mathcal{S}$)
  shows $\mathcal{S}$ = $\mathcal{S}_{sb}$ $\wedge$ m = m_{sb} $\wedge$ length ts = length ts_{sb} $\wedge$
    ($\forall$ i < length ts_{sb}.
      let (p, is, j, sb, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$) = ts_{sb}!i
      in ts!i = (p, is, j, (), $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$))
⟨*proof*⟩

lemma sim-config-emptyI:
  assumes empty:
  $\forall$ i p is xs sb $\mathcal{D}$ $\mathcal{O}$ $\mathcal{R}$. i < length ts_{sb} $\longrightarrow$ ts_{sb}!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$\longrightarrow$ sb=[]
  assumes leq: length ts = length ts_{sb}
  assumes ts: ($\forall$ i < length ts_{sb}.
      let (p, is, j, sb, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$) = ts_{sb}!i
      in ts!i = (p, is, j, (), $\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$))
  shows (ts_{sb},m_{sb},$\mathcal{S}_{sb}$) $\sim$ (ts,m_{sb},$\mathcal{S}_{sb}$)
⟨*proof*⟩
lemma mem-eq-un-eq: ⟦length ts′=length ts; $\forall$ i< length ts′. P (ts′!i) = Q (ts!i) ⟧ $\Longrightarrow$ ($\bigcup$x∈set ts′. P x) = ($\bigcup$x∈set ts. Q x)
⟨*proof*⟩

lemma (**in** program) trace-to-steps:
assumes trace: trace c 0 k
shows steps: c 0 $\Rightarrow_d{}^*$ c k
⟨*proof*⟩

lemma (**in** program) safe-reach-to-safe-reach-upto:
  assumes safe-reach: safe-reach-direct safe c_0
  shows safe-reach-upto n safe c_0
⟨*proof*⟩

lemma (**in** program-progress) safe-free-flowing-implies-safe-delayed′:
  assumes init: $initial_{sb}$ ts_{sb} $\mathcal{S}_{sb}$
  assumes sim: (ts_{sb},m_{sb},$\mathcal{S}_{sb}$) $\sim$ (ts,m,$\mathcal{S}$)
  assumes safe-reach-ff: safe-reach-direct safe-free-flowing (ts,m,$\mathcal{S}$)
  shows safe-reach-direct safe-delayed (ts,m,$\mathcal{S}$)
⟨*proof*⟩

lemma map-onws-sb-owned:$\bigwedge$j. j < length ts $\Longrightarrow$ map $\mathcal{O}$-sb ts ! j = ($\mathcal{O}_j$,sb_j) $\Longrightarrow$ map owned ts ! j = $\mathcal{O}_j$
⟨*proof*⟩

lemma map-onws-sb-owned′:$\bigwedge$j. j < length ts $\Longrightarrow$ $\mathcal{O}$-sb (ts ! j) = ($\mathcal{O}_j$,sb_j) $\Longrightarrow$ owned (ts ! j) = $\mathcal{O}_j$
⟨*proof*⟩

lemma read-only-read-acquired-unforwarded-acquire-witness:
  $\bigwedge\mathcal{S}$ $\mathcal{O}$ X.⟦non-volatile-owned-or-read-only True $\mathcal{S}$ $\mathcal{O}$ sb;
 sharing-consistent $\mathcal{S}$ $\mathcal{O}$ sb; a $\notin$ read-only $\mathcal{S}$; a $\notin$ $\mathcal{O}$;
 a $\in$ unforwarded-non-volatile-reads sb X⟧
$\Longrightarrow$($\exists$ sop a′ v ys zs A L R W.
   sb = ys @ $Write_{sb}$ True a′ sop v A L R W # zs $\wedge$
   a $\in$ A $\wedge$ a $\notin$ outstanding-refs is-$Write_{sb}$ ys $\wedge$ a′ $\neq$ a) $\vee$

($\exists$A L R W ys zs. sb = ys @ Ghost$_{sb}$ A L R W# zs $\wedge$ a $\in$ A $\wedge$ a $\notin$ outstanding-refs is-Write$_{sb}$ ys)
⟨*proof*⟩

lemma release-shared-exchange-weak:
assumes shared-eq: $\forall$a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb. ($\mathcal{S}'$::shared) a = $\mathcal{S}$ a
assumes consis: weak-sharing-consistent $\mathcal{O}$ sb
shows release sb (dom $\mathcal{S}'$) $\mathcal{R}$ = release sb (dom $\mathcal{S}$) $\mathcal{R}$
⟨*proof*⟩

lemma read-only-share-all-shared: $\bigwedge\mathcal{S}$. ⟦ a $\in$ read-only (share sb $\mathcal{S}$)⟧
$\Longrightarrow$ a $\in$ read-only $\mathcal{S}$ $\cup$ all-shared sb
⟨*proof*⟩

lemma read-only-shared-all-until-volatile-write-subset$'$:
$\bigwedge\mathcal{S}$.
read-only (share-all-until-volatile-write ts $\mathcal{S}$) $\subseteq$
read-only $\mathcal{S}$ $\cup$ ($\bigcup$((λ(-, -, -, sb, -, - ,-). all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)) ‘ set ts))
⟨*proof*⟩

lemma read-only-share-acquired-all-shared:
$\bigwedge\mathcal{O}$ $\mathcal{S}$. weak-sharing-consistent $\mathcal{O}$ sb $\Longrightarrow$ $\mathcal{O}$ $\cap$ read-only $\mathcal{S}$ = {} $\Longrightarrow$
a $\in$ read-only (share sb $\mathcal{S}$) $\Longrightarrow$ a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb $\Longrightarrow$ a $\in$ all-shared sb
⟨*proof*⟩

lemma read-only-share-unowned$'$: $\bigwedge\mathcal{O}$ $\mathcal{S}$.
⟦weak-sharing-consistent $\mathcal{O}$ sb; $\mathcal{O}$ $\cap$ read-only $\mathcal{S}$ = {}; a $\notin$ $\mathcal{O}$ $\cup$ all-acquired sb; a $\in$ read-only $\mathcal{S}$⟧
$\Longrightarrow$ a $\in$ read-only (share sb $\mathcal{S}$)
⟨*proof*⟩

lemma release-False-mono:
$\bigwedge$S $\mathcal{R}$. $\mathcal{R}$ a = Some False $\Longrightarrow$ outstanding-refs is-volatile-Write$_{sb}$ sb = {} $\Longrightarrow$
release sb S $\mathcal{R}$ a = Some False
⟨*proof*⟩

lemma release-False-mono-take:
$\bigwedge$S $\mathcal{R}$. $\mathcal{R}$ a = Some False $\Longrightarrow$ release (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) S $\mathcal{R}$ a = Some False
⟨*proof*⟩

lemma shared-switch:
$\bigwedge\mathcal{S}$ $\mathcal{O}$. ⟦weak-sharing-consistent $\mathcal{O}$ sb; read-only $\mathcal{S}$ $\cap$ $\mathcal{O}$ = {};
$\mathcal{S}$ a $\neq$ Some False; share sb $\mathcal{S}$ a = Some False⟧
$\Longrightarrow$ a $\in$ $\mathcal{O}$ $\cup$ all-acquired sb
⟨*proof*⟩

lemma shared-switch-release-False:
$\bigwedge\mathcal{S}$ $\mathcal{R}$. ⟦
outstanding-refs is-volatile-Write$_{sb}$ sb = {};
a $\notin$ dom $\mathcal{S}$;
a $\in$ dom (share sb $\mathcal{S}$)⟧
$\Longrightarrow$
release sb (dom $\mathcal{S}$) $\mathcal{R}$ a = Some False
⟨*proof*⟩

lemma release-not-unshared-no-write:
$\bigwedge\mathcal{S}\ \mathcal{R}.\ [\![$
outstanding-refs is-volatile-Write$_{sb}$ sb $= \{\}$;
non-volatile-writes-unshared $\mathcal{S}$ sb;
release sb (dom $\mathcal{S}$) $\mathcal{R}$ a $\neq$ Some False;
a $\in$ dom (share sb $\mathcal{S}$)$]\!]$
$\Longrightarrow$
a $\notin$ outstanding-refs is-non-volatile-Write$_{sb}$ sb
$\langle proof \rangle$

corollary release-not-unshared-no-write-take:
assumes nvw: non-volatile-writes-unshared $\mathcal{S}$ (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)
assumes rel: release (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) (dom $\mathcal{S}$) $\mathcal{R}$ a $\neq$ Some False
assumes a-in: a $\in$ dom (share (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\mathcal{S}$)
shows
a $\notin$ outstanding-refs is-non-volatile-Write$_{sb}$ (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)
$\langle proof \rangle$

lemma read-only-unacquired-share$'$:
$\bigwedge$S $\mathcal{O}$. $[\![\mathcal{O} \cap$ read-only S $= \{\}$; weak-sharing-consistent $\mathcal{O}$ sb; a $\in$ read-only S;
a $\notin$ all-shared sb; a $\notin$ acquired True sb $\mathcal{O}$ $]\!]$
$\Longrightarrow$ a $\in$ read-only (share sb S)
$\langle proof \rangle$

lemma read-only-share-all-until-volatile-write-unacquired$'$:
$\bigwedge\mathcal{S}$. $[\![$ownership-distinct ts; read-only-unowned $\mathcal{S}$ ts; weak-sharing-consis ts;
$\forall$i $<$ length ts. (let (-,-,-,sb,-,$\mathcal{O}$,$\mathcal{R}$) = ts!i in
a $\notin$ acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb) $\mathcal{O}$ $\wedge$
a $\notin$ all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb
));
a $\in$ read-only $\mathcal{S}]\!]$
$\Longrightarrow$ a $\in$ read-only (share-all-until-volatile-write ts $\mathcal{S}$)
$\langle proof \rangle$

lemma not-shared-not-acquired-switch:
$\bigwedge$X Y. $[\![$a $\notin$ all-shared sb; a $\notin$ X; a $\notin$ acquired True sb X; a $\notin$ Y$]\!]$ $\Longrightarrow$ a $\notin$ acquired True sb Y
$\langle proof \rangle$

lemma read-only-share-all-acquired-in$'$:
$\bigwedge$S $\mathcal{O}$. $[\![\mathcal{O} \cap$ read-only S $= \{\}$; weak-sharing-consistent $\mathcal{O}$ sb; a $\in$ read-only (share sb S)$]\!]$
$\Longrightarrow$ a $\in$ read-only (share sb Map.empty) $\vee$ (a $\in$ read-only S $\wedge$ a $\notin$ acquired True sb $\mathcal{O}$ $\wedge$ a $\notin$ all-shared sb)
$\langle proof \rangle$

lemma in-read-only-share-all-until-volatile-write$'$:
assumes dist: ownership-distinct ts
assumes consis: sharing-consis $\mathcal{S}$ ts
assumes ro-unowned: read-only-unowned $\mathcal{S}$ ts
assumes i-bound: i $<$ length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes a-unacquired-others: $\forall$j $<$ length ts. i$\neq$j $\longrightarrow$
(let (-,-,-,sb$_j$,-,$\mathcal{O}$,-) = ts!j in
a $\notin$ acquired True (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$) $\mathcal{O}$ $\wedge$
a $\notin$ all-shared (takeWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb$_j$))
assumes a-ro-share: a $\in$ read-only (share sb $\mathcal{S}$)
shows a $\in$ read-only (share (dropWhile (Not $\circ$ is-volatile-Write$_{sb}$) sb)

(share-all-until-volatile-write ts $\mathcal{S}$))
⟨*proof*⟩

lemma all-acquired-unshared-acquired:
$\bigwedge\mathcal{O}$. a ∈ all-acquired sb ==> a ∉ all-shared sb ==> a ∈ acquired True sb $\mathcal{O}$
⟨*proof*⟩

lemma safe-RMW-common:
assumes safe: $\mathcal{O}$s,$\mathcal{R}$s,i⊢ (RMW a t (D,f) cond ret A L R W# is, j, m, $\mathcal{D}$, $\mathcal{O}$, $\mathcal{S}$)$\surd$
shows (a ∈ $\mathcal{O}$ ∨ a ∈ dom $\mathcal{S}$) ∧ (∀ j < length $\mathcal{O}$s. i≠j ⟶ ($\mathcal{R}$s!j) a ≠ Some False)
⟨*proof*⟩

lemma acquired-reads-all-acquired′: $\bigwedge\mathcal{O}$.
acquired-reads True sb $\mathcal{O}$ ⊆ acquired True sb $\mathcal{O}$ ∪ all-shared sb
⟨*proof*⟩

lemma release-all-shared-exchange:
$\bigwedge\mathcal{R}$ S′ S. ∀ a ∈ all-shared sb. (a ∈ S′) = (a ∈ S) ⟹ release sb S′ $\mathcal{R}$ = release sb S $\mathcal{R}$
⟨*proof*⟩

lemma release-append-Prog$_{sb}$:
$\bigwedge$S $\mathcal{R}$. (release (takeWhile (Not ∘ is-volatile-Write$_{sb}$) (sb @ [Prog$_{sb}$ p_1 p_2 mis])) S $\mathcal{R}$) =
(release (takeWhile (Not ∘ is-volatile-Write$_{sb}$) sb) S $\mathcal{R}$)
⟨*proof*⟩

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},$\mathcal{S}_{sb}$) $\Rightarrow_{sbh}$ (ts_{sb}′,m_{sb}′,$\mathcal{S}_{sb}$′)
assumes valid-own: valid-ownership $\mathcal{S}_{sb}$ 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 $\mathcal{S}_{sb}$ ts_{sb}
assumes tmps-distinct: tmps-distinct ts_{sb}
assumes valid-sops: valid-sops ts_{sb}
assumes valid-dd: valid-data-dependency ts_{sb}
assumes load-tmps-fresh: load-tmps-fresh ts_{sb}
assumes enough-flushs: enough-flushs ts_{sb}
assumes valid-program-history: valid-program-history ts_{sb}
assumes valid: valid ts_{sb}
assumes sim: (ts_{sb},m_{sb},$\mathcal{S}_{sb}$) ∼ (ts,m,$\mathcal{S}$)
assumes safe-reach: safe-reach-direct safe-delayed (ts,m,$\mathcal{S}$)
shows valid-ownership $\mathcal{S}_{sb}$′ ts_{sb}′ ∧ valid-reads m_{sb}′ ts_{sb}′ ∧ valid-history program-step ts_{sb}′ ∧
valid-sharing $\mathcal{S}_{sb}$′ ts_{sb}′ ∧ tmps-distinct ts_{sb}′ ∧ valid-data-dependency ts_{sb}′ ∧
valid-sops ts_{sb}′ ∧ load-tmps-fresh ts_{sb}′ ∧ enough-flushs ts_{sb}′ ∧
valid-program-history ts_{sb}′ ∧ valid ts_{sb}′ ∧
(∃ ts′ $\mathcal{S}$′ m′. (ts,m,$\mathcal{S}$) $\Rightarrow_d^*$ (ts′,m′,$\mathcal{S}$′) ∧
(ts_{sb}′,m_{sb}′,$\mathcal{S}_{sb}$′) ∼ (ts′,m′,$\mathcal{S}$′))

$\langle proof \rangle$

theorem (**in** xvalid-program) concurrent-direct-steps-simulates-store-buffer-history-steps:
assumes step-sb: $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \Rightarrow_{sbh}^* (ts_{sb}',m_{sb}',\mathcal{S}_{sb}')$
assumes valid-own: valid-ownership $\mathcal{S}_{sb}$ 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 $\mathcal{S}_{sb}$ ts_{sb}
assumes tmps-distinct: tmps-distinct ts_{sb}
assumes valid-sops: valid-sops ts_{sb}
assumes valid-dd: valid-data-dependency ts_{sb}
assumes load-tmps-fresh: load-tmps-fresh ts_{sb}
assumes enough-flushs: enough-flushs ts_{sb}
assumes valid-program-history: valid-program-history ts_{sb}
assumes valid: valid ts_{sb}
shows $\bigwedge$ts $\mathcal{S}$ m. $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \sim (ts,m,\mathcal{S}) \Longrightarrow$ safe-reach-direct safe-delayed $(ts,m,\mathcal{S})$
$\Longrightarrow$
valid-ownership $\mathcal{S}_{sb}'$ ts_{sb}' $\wedge$ valid-reads m_{sb}' ts_{sb}' $\wedge$ valid-history program-step ts_{sb}'
$\wedge$
valid-sharing $\mathcal{S}_{sb}'$ ts_{sb}' $\wedge$ tmps-distinct ts_{sb}' $\wedge$ valid-data-dependency ts_{sb}' $\wedge$
valid-sops ts_{sb}' $\wedge$ load-tmps-fresh ts_{sb}' $\wedge$ enough-flushs ts_{sb}' $\wedge$
valid-program-history ts_{sb}' $\wedge$ valid ts_{sb}' $\wedge$
$(\exists ts'\ m'\ \mathcal{S}'.\ (ts,m,\mathcal{S}) \Rightarrow_d^* (ts',m',\mathcal{S}') \wedge (ts_{sb}',m_{sb}',\mathcal{S}_{sb}') \sim (ts',m',\mathcal{S}'))$
$\langle proof \rangle$

sublocale $initial_{sb} \subseteq$ tmps-distinct $\langle proof \rangle$
locale xvalid-program-progress = program-progress + xvalid-program

theorem (**in** xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-history-execution:
assumes exec-sb: $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \Rightarrow_{sbh}^* (ts_{sb}',m_{sb}',\mathcal{S}_{sb}')$
assumes init: $initial_{sb}$ ts_{sb} $\mathcal{S}_{sb}$
assumes valid: valid ts_{sb}
assumes sim: $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \sim (ts,m,\mathcal{S})$
assumes safe: safe-reach-direct safe-free-flowing $(ts,m,\mathcal{S})$
shows $\exists ts'\ m'\ \mathcal{S}'.\ (ts,m,\mathcal{S}) \Rightarrow_d^* (ts',m',\mathcal{S}') \wedge$
$(ts_{sb}',m_{sb}',\mathcal{S}_{sb}') \sim (ts',m',\mathcal{S}')$
$\langle proof \rangle$

lemma filter-is-$Write_{sb}$-Cons-$Write_{sb}$: filter is-$Write_{sb}$ xs = $Write_{sb}$ volatile a sop v A L R W#ys
$\Longrightarrow \exists$rs rws. ($\forall$r $\in$ set rs. is-$Read_{sb}$ r $\vee$ is-$Prog_{sb}$ r $\vee$ is-$Ghost_{sb}$ r) $\wedge$
xs=rs@$Write_{sb}$ volatile a sop v A L R W#rws $\wedge$ ys=filter is-$Write_{sb}$ rws
$\langle proof \rangle$

lemma filter-is-Write$_{sb}$-empty: filter is-Write$_{sb}$ xs = []
$\Longrightarrow$ ($\forall$ r $\in$ set xs. is-Read$_{sb}$ r $\vee$ is-Prog$_{sb}$ r $\vee$ is-Ghost$_{sb}$ r)
⟨*proof*⟩

lemma flush-reads-program: $\bigwedge\mathcal{O}\ \mathcal{S}\ \mathcal{R}$.
$\forall$ r $\in$ set sb. is-Read$_{sb}$ r $\vee$ is-Prog$_{sb}$ r $\vee$ is-Ghost$_{sb}$ r $\Longrightarrow$
$\exists\,\mathcal{O}'\ \mathcal{R}'\ \mathcal{S}'$. (m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{f}^{*}$ (m,[],$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
⟨*proof*⟩

lemma flush-progress: $\exists$ m′ $\mathcal{O}'\ \mathcal{S}'\ \mathcal{R}'$. (m,r#sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{f}$ (m′,sb,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
⟨*proof*⟩

lemma flush-empty:
assumes steps: (m, sb,$\mathcal{O}$,$\mathcal{R}$, $\mathcal{S}$) $\rightarrow_{f}^{*}$ (m′, sb′,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
shows sb=[] $\Longrightarrow$ m′=m $\wedge$ sb′=[] $\wedge$ $\mathcal{O}'$=$\mathcal{O}$ $\wedge$ $\mathcal{R}'$=$\mathcal{R}$ $\wedge$ $\mathcal{S}'$=$\mathcal{S}$
⟨*proof*⟩

lemma flush-append:
assumes steps: (m, sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{f}^{*}$ (m′, sb′,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
shows $\bigwedge$xs. (m, sb@xs,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{f}^{*}$ (m′, sb′@xs,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
⟨*proof*⟩

lemmas store-buffer-step-induct =
store-buffer-step.induct [split-format (complete),
consumes 1, case-names SBWrite$_{sb}$]
theorem flush-simulates-filter-writes:
assumes step: (m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{w}$ (m′,sb′,$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
shows $\bigwedge$sb$_{h}$ $\mathcal{O}_{h}$ $\mathcal{R}_{h}$ $\mathcal{S}_{h}$. sb=filter is-Write$_{sb}$ sb$_{h}$
$\Longrightarrow$
$\exists$ sb$_{h}$′ $\mathcal{O}_{h}'$ $\mathcal{R}_{h}'$ $\mathcal{S}_{h}'$. (m,sb$_{h}$,$\mathcal{O}_{h}$,$\mathcal{R}_{h}$,$\mathcal{S}_{h}$) $\rightarrow_{f}^{*}$ (m′,sb$_{h}$′,$\mathcal{O}_{h}'$,$\mathcal{R}_{h}'$,$\mathcal{S}_{h}'$) $\wedge$
sb′=filter is-Write$_{sb}$ sb$_{h}$′ $\wedge$ (sb′=[] $\longrightarrow$ sb$_{h}$′=[])
⟨*proof*⟩

lemma bufferd-val-filter-is-Write$_{sb}$-eq-ext:
buffered-val (filter is-Write$_{sb}$ sb) a = buffered-val sb a
⟨*proof*⟩

lemma bufferd-val-filter-is-Write$_{sb}$-eq:
buffered-val (filter is-Write$_{sb}$ sb) = buffered-val sb
⟨*proof*⟩

lemma outstanding-refs-is-volatile-Write$_{sb}$-filter-writes:
outstanding-refs is-volatile-Write$_{sb}$ (filter is-Write$_{sb}$ xs) =
outstanding-refs is-volatile-Write$_{sb}$ xs
⟨*proof*⟩

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: $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{sb}} (ts',m',\mathcal{S}')$
  assumes sim: $ts \sim_{\mathsf{h}} ts_{\mathsf{h}}$
  shows $\exists ts_{\mathsf{h}}'\ \mathcal{S}_{\mathsf{h}}'.\ (ts_{\mathsf{h}},m,\mathcal{S}_{\mathsf{h}}) \Rightarrow_{\mathsf{sbh}}{}^{*} (ts_{\mathsf{h}}',m',\mathcal{S}_{\mathsf{h}}') \wedge ts' \sim_{\mathsf{h}} ts_{\mathsf{h}}'$
⟨*proof*⟩

theorem (**in** valid-program) concurrent-history-steps-simulates-store-buffer-steps:
  assumes step-sb: $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{sb}}{}^{*} (ts',m',\mathcal{S}')$
  shows $\bigwedge ts_{\mathsf{h}}\ \mathcal{S}_{\mathsf{h}}.\ ts \sim_{\mathsf{h}} ts_{\mathsf{h}} \Longrightarrow \exists ts_{\mathsf{h}}'\ \mathcal{S}_{\mathsf{h}}'.\ (ts_{\mathsf{h}},m,\mathcal{S}_{\mathsf{h}}) \Rightarrow_{\mathsf{sbh}}{}^{*} (ts_{\mathsf{h}}',m',\mathcal{S}_{\mathsf{h}}') \wedge ts' \sim_{\mathsf{h}} ts_{\mathsf{h}}'$
⟨*proof*⟩

theorem (**in** xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-execution:
assumes exec-sb: $(ts_{\mathsf{sb}},m_{\mathsf{sb}},x) \Rightarrow_{\mathsf{sb}}{}^{*} (ts_{\mathsf{sb}}',m_{\mathsf{sb}}',x')$
assumes init: $\text{initial}_{\mathsf{sb}}\ ts_{\mathsf{sb}}\ \mathcal{S}_{\mathsf{sb}}$
assumes valid: valid ts_{sb}
assumes sim: $(ts_{\mathsf{sb}},m_{\mathsf{sb}},\mathcal{S}_{\mathsf{sb}}) \sim (ts,m,\mathcal{S})$
assumes safe: safe-reach-direct safe-free-flowing $(ts,m,\mathcal{S})$
shows $\exists ts_{\mathsf{h}}'\ \mathcal{S}_{\mathsf{h}}'\ ts'\ m'\ \mathcal{S}'.$
      $(ts_{\mathsf{sb}},m_{\mathsf{sb}},\mathcal{S}_{\mathsf{sb}}) \Rightarrow_{\mathsf{sbh}}{}^{*} (ts_{\mathsf{h}}',m_{\mathsf{sb}}',\mathcal{S}_{\mathsf{h}}') \wedge$
        $ts_{\mathsf{sb}}' \sim_{\mathsf{h}} ts_{\mathsf{h}}' \wedge$
      $(ts,m,\mathcal{S}) \Rightarrow_{\mathsf{d}}{}^{*} (ts',m',\mathcal{S}') \wedge$
        $(ts_{\mathsf{h}}',m_{\mathsf{sb}}',\mathcal{S}_{\mathsf{h}}') \sim (ts',m',\mathcal{S}')$
⟨*proof*⟩

inductive sim-direct-config::
  ('p,'p store-buffer,'dirty,'owns,'rels) thread-config list ⇒ ('p,unit,bool,'owns','rels') thread-config list ⇒ bool
  (‹- $\sim_{\mathsf{d}}$ - › [60,60] 100)
where
  ⟦length ts = length ts_{d};
   $\forall$ i < length ts.
     $(\exists \mathcal{O}'\ \mathcal{D}'\ \mathcal{R}'.$
      let $(p,is,j,sb,\mathcal{D},\mathcal{O},\mathcal{R}) = ts_{\mathsf{d}}!i$ in
        $ts!i=(p,is,j,[],\mathcal{D}',\mathcal{O}',\mathcal{R}'))$
  ⟧
   $\Longrightarrow$
   $ts \sim_{\mathsf{d}} ts_{\mathsf{d}}$

lemma empty-sb-sims:
assumes empty:
  $\forall$ i p is xs sb $\mathcal{D}\ \mathcal{O}\ \mathcal{R}$. i < length $ts_{\mathsf{sb}} \longrightarrow ts_{\mathsf{sb}}!i=(p,is,xs,sb,\mathcal{D},\mathcal{O},\mathcal{R}) \longrightarrow sb=[]$
assumes sim-h: $ts_{\mathsf{sb}} \sim_{\mathsf{h}} ts_{\mathsf{h}}$
assumes sim-d: $(ts_{\mathsf{h}},m_{\mathsf{h}},\mathcal{S}_{\mathsf{h}}) \sim (ts,m,\mathcal{S})$

shows $ts_{sb} \sim_d ts \wedge m_h=m \wedge \text{length } ts_{sb} = \text{length } ts$
⟨*proof*⟩

lemma empty-d-sims:
assumes sim: $ts_{sb} \sim_d ts$
shows $\exists ts_h.\ ts_{sb} \sim_h ts_h \wedge (ts_h,m,\mathcal{S}) \sim (ts,m,\mathcal{S})$
⟨*proof*⟩

theorem (**in** xvalid-program-progress) concurrent-direct-execution-simulates-store-buffer-execution-empty:
assumes exec-sb: $(ts_{sb},m_{sb},x) \Rightarrow_{sb}^* (ts_{sb}',m_{sb}',x')$
assumes init: $\text{initial}_{sb}\ ts_{sb}\ \mathcal{S}_{sb}$
assumes valid: valid ts_{sb}
assumes empty:
$\forall i\ p\ is\ xs\ sb\ \mathcal{D}\ \mathcal{O}\ \mathcal{R}.\ i < \text{length } ts_{sb}' \longrightarrow ts_{sb}'!i=(p,is,xs,sb,\mathcal{D},\mathcal{O},\mathcal{R}) \longrightarrow sb=[]$
assumes sim: $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \sim (ts,m,\mathcal{S})$
assumes safe: safe-reach-direct safe-free-flowing $(ts,m,\mathcal{S})$
shows $\exists ts'\ \mathcal{S}'.$
$(ts,m,\mathcal{S}) \Rightarrow_d^* (ts',m_{sb}',\mathcal{S}') \wedge ts_{sb}' \sim_d ts'$
⟨*proof*⟩

locale initial_d = simple-ownership-distinct + read-only-unowned + unowned-shared +
fixes valid
assumes empty-is: $[\![i < \text{length } ts;\ ts!i=(p,is,xs,sb,\mathcal{D},\mathcal{O},\mathcal{R})]\!] \Longrightarrow is=[]$
assumes empty-rels: $[\![i < \text{length } ts;\ ts!i=(p,is,xs,sb,\mathcal{D},\mathcal{O},\mathcal{R})]\!] \Longrightarrow \mathcal{R}=\text{Map.empty}$
assumes valid-init: valid (map $(\lambda(p,is, j,sb,\mathcal{D}, \mathcal{O},\mathcal{R}).\ (p,is, j,[],\mathcal{D}, \mathcal{O},\mathcal{R}))$ ts)

locale empty-store-buffers =
fixes ts::('p,'p store-buffer,bool,owns,rels) thread-config list
assumes empty-sb: $[\![i < \text{length } ts;\ ts!i=(p,is,xs,sb,\mathcal{D},\mathcal{O},\mathcal{R})]\!] \Longrightarrow sb=[]$

lemma initial-d-sb:
assumes init: initial_d ts $\mathcal{S}$ valid
shows initial_{sb} (map $(\lambda(p,is, j,sb,\mathcal{D}, \mathcal{O},\mathcal{R}).\ (p,is, j,[],\mathcal{D}, \mathcal{O},\mathcal{R}))$ ts) $\mathcal{S}$
(**is** initial_{sb} ?map $\mathcal{S}$)
⟨*proof*⟩

theorem (**in** xvalid-program-progress) store-buffer-execution-result-sequential-consistent:
assumes exec-sb: $(ts_{sb},m,x) \Rightarrow_{sb}^* (ts_{sb}',m',x')$
assumes empty': empty-store-buffers ts_{sb}'
assumes sim: $ts_{sb} \sim_d ts$
assumes init: initial_d ts $\mathcal{S}$ valid
assumes safe: safe-reach-direct safe-free-flowing $(ts,m,\mathcal{S})$
shows $\exists ts'\ \mathcal{S}'.$
$(ts,m,\mathcal{S}) \Rightarrow_d^* (ts',m',\mathcal{S}') \wedge ts_{sb}' \sim_d ts'$
⟨*proof*⟩

locale initial_v = simple-ownership-distinct + read-only-unowned + unowned-shared +
fixes valid

assumes empty-is: $[\![$i < length ts; ts!i=(p,is,xs,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)$]\!] \Longrightarrow$ is=[]
assumes valid-init: valid (map (λ(p,is, j,sb,$\mathcal{D}$, $\mathcal{O}$,$\mathcal{R}$). (p,is, j,[],$\mathcal{D}$, $\mathcal{O}$,Map.empty)) ts)

theorem (**in** xvalid-program-progress) store-buffer-execution-result-sequential-consistent$'$:
assumes exec-sb: (ts_{sb},m,x) $\Rightarrow_{sb}{}^*$ ($ts_{sb}{}'$,m$'$,x$'$)
assumes empty$'$: empty-store-buffers $ts_{sb}{}'$
assumes sim: $ts_{sb} \sim_d$ ts
assumes init: $initial_v$ ts $\mathcal{S}$ valid
assumes safe: safe-reach-virtual safe-free-flowing (ts,m,$\mathcal{S}$)
shows $\exists$ts$'$ $\mathcal{S}'$.
(ts,m,$\mathcal{S}$) $\Rightarrow_v{}^*$ (ts$'$,m$'$,$\mathcal{S}'$) $\wedge$ $ts_{sb}{}' \sim_d$ ts$'$
$\langle$*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},$\mathcal{S}_{sb}$) $\Rightarrow_{sb}$ ($ts_{sb}{}'$,$m_{sb}{}'$,$\mathcal{S}_{sb}{}'$)
assumes sim-h: $ts_{sb} \sim_h ts_{sbh}$
assumes sim: (ts_{sbh},m_{sb},$\mathcal{S}_{sbh}$) $\sim$ (ts,m,$\mathcal{S}$)
assumes valid-own: valid-ownership $\mathcal{S}_{sbh}$ 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 $\mathcal{S}_{sbh}$ ts_{sbh}
assumes tmps-distinct: tmps-distinct ts_{sbh}
assumes valid-sops: valid-sops ts_{sbh}
assumes valid-dd: valid-data-dependency ts_{sbh}
assumes load-tmps-fresh: load-tmps-fresh ts_{sbh}
assumes enough-flushs: enough-flushs ts_{sbh}
assumes valid-program-history: valid-program-history ts_{sbh}
assumes valid: valid ts_{sbh}
assumes safe-reach: safe-reach-direct safe-delayed (ts,m,$\mathcal{S}$)
shows $\exists ts_{sbh}{}'$ $\mathcal{S}_{sbh}{}'$.
(ts_{sbh},m_{sb},$\mathcal{S}_{sbh}$) $\Rightarrow_{sbh}{}^*$ ($ts_{sbh}{}'$,$m_{sb}{}'$,$\mathcal{S}_{sbh}{}'$) $\wedge$ $ts_{sb}{}' \sim_h ts_{sbh}{}'$ $\wedge$
valid-ownership $\mathcal{S}_{sbh}{}'$ $ts_{sbh}{}'$ $\wedge$ valid-reads $m_{sb}{}'$ $ts_{sbh}{}'$ $\wedge$
valid-history program-step $ts_{sbh}{}'$ $\wedge$
valid-sharing $\mathcal{S}_{sbh}{}'$ $ts_{sbh}{}'$ $\wedge$ tmps-distinct $ts_{sbh}{}'$ $\wedge$ valid-data-dependency $ts_{sbh}{}'$ $\wedge$
valid-sops $ts_{sbh}{}'$ $\wedge$ load-tmps-fresh $ts_{sbh}{}'$ $\wedge$ enough-flushs $ts_{sbh}{}'$ $\wedge$
valid-program-history $ts_{sbh}{}'$ $\wedge$ valid $ts_{sbh}{}'$ $\wedge$
($\exists$ts$'$ $\mathcal{S}'$ m$'$. (ts,m,$\mathcal{S}$) $\Rightarrow_d{}^*$ (ts$'$,m$'$,$\mathcal{S}'$) $\wedge$
($ts_{sbh}{}'$,$m_{sb}{}'$,$\mathcal{S}_{sbh}{}'$) $\sim$ (ts$'$,m$'$,$\mathcal{S}'$))
$\langle$*proof*$\rangle$

lemma conj-commI: P $\wedge$ Q $\Longrightarrow$ Q $\wedge$ P
$\langle$*proof*$\rangle$

lemma def-to-eq: P = Q $\Longrightarrow$ P $\equiv$ Q
⟨*proof*⟩

context xvalid-program
begin

definition
invariant ts $\mathcal{S}$ m $\equiv$
valid-ownership $\mathcal{S}$ ts $\wedge$ valid-reads m ts $\wedge$ valid-history program-step ts $\wedge$
valid-sharing $\mathcal{S}$ ts $\wedge$ tmps-distinct ts $\wedge$ valid-data-dependency ts $\wedge$
valid-sops ts $\wedge$ load-tmps-fresh ts $\wedge$ enough-flushs ts $\wedge$ valid-program-history ts $\wedge$
valid ts

definition ownership-inv $\equiv$ valid-ownership
definition sharing-inv $\equiv$ valid-sharing
definition temporaries-inv ts $\equiv$ tmps-distinct ts $\wedge$ load-tmps-fresh ts
definition history-inv ts m $\equiv$ valid-history program-step ts $\wedge$ valid-program-history ts $\wedge$ valid-reads m ts
definition data-dependency-inv ts $\equiv$ valid-data-dependency ts $\wedge$ load-tmps-fresh ts $\wedge$ valid-sops ts
definition barrier-inv $\equiv$ enough-flushs

lemma invariant-grouped-def: invariant ts $\mathcal{S}$ m $\equiv$
ownership-inv $\mathcal{S}$ ts $\wedge$ sharing-inv $\mathcal{S}$ ts $\wedge$ temporaries-inv ts $\wedge$ data-dependency-inv ts $\wedge$ history-inv ts m $\wedge$ barrier-inv ts $\wedge$ valid ts
⟨*proof*⟩

theorem (**in** xvalid-program) simulation′:
assumes step-sb: $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \Rightarrow_{sbh} (ts_{sb}',m_{sb}',\mathcal{S}_{sb}')$
assumes sim: $(ts_{sb},m_{sb},\mathcal{S}_{sb}) \sim (ts,m,\mathcal{S})$
assumes inv: invariant ts_{sb} $\mathcal{S}_{sb}$ m_{sb}
assumes safe-reach: safe-reach-direct safe-delayed (ts,m,$\mathcal{S}$)
shows invariant ts_{sb}' $\mathcal{S}_{sb}'$ m_{sb}' $\wedge$
$(\exists ts'\ \mathcal{S}'\ m'.\ (ts,m,\mathcal{S}) \Rightarrow_{d}^{*} (ts',m',\mathcal{S}') \wedge (ts_{sb}',m_{sb}',\mathcal{S}_{sb}') \sim (ts',m',\mathcal{S}'))$
⟨*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

| Binop val ⇒ val ⇒ val expr expr

datatype stmt =
Skip
| Assign bool expr expr tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns
| CAS expr expr expr tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns
| Seq stmt stmt
| Cond expr stmt stmt
| While expr stmt

| SGhost tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns tmps ⇒ owns
| SFence

primrec used-tmps:: expr ⇒ nat — number of temporaries used
where
used-tmps (Const v) = 0
| used-tmps (Mem volatile addr) = 1
| used-tmps (Tmp sop) = 0
| used-tmps (Unop f e) = used-tmps e
| used-tmps (Binop f e_1 e_2) = used-tmps e_1 + used-tmps e_2

primrec issue-expr:: tmp ⇒ expr ⇒ instr list — load operations
where
issue-expr t (Const v) = []
|issue-expr t (Mem volatile a) = [Read volatile a t]
|issue-expr t (Tmp sop) = []
|issue-expr t (Unop f e) = issue-expr t e
|issue-expr t (Binop f e_1 e_2) = issue-expr t e_1 @ issue-expr (t + (used-tmps e_1)) e_2

primrec eval-expr:: tmp ⇒ expr ⇒ sop — calculate result
where
eval-expr t (Const v) = ({},λj. v)
|eval-expr t (Mem volatile a) = ({t},λj. the (j 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,λj. f (f_e j)))
|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 ($D_1 \cup D_2$,λj. f (f_1 j) (f_2 j)))

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) = (($\forall t' \in$ fst sop. $t' < t$) ∧ valid-sop sop)

|valid-sops-expr t (Unop f e) = valid-sops-expr t e
|valid-sops-expr t (Binop f e_1 e_2) = (valid-sops-expr t e_1 ∧ valid-sops-expr t e_2)

primrec valid-sops-stmt:: nat ⇒ stmt ⇒ bool
where
valid-sops-stmt t Skip = True
|valid-sops-stmt t (Assign volatile a e A L R W) = (valid-sops-expr t a ∧ valid-sops-expr t e)
|valid-sops-stmt t (CAS a c_e s_e A L R W) = (valid-sops-expr t a ∧ valid-sops-expr t c_e ∧
valid-sops-expr t s_e)
|valid-sops-stmt t (Seq s_1 s_2) = (valid-sops-stmt t s_1 ∧ valid-sops-stmt t s_2)
|valid-sops-stmt t (Cond e s_1 s_2) = (valid-sops-expr t e ∧ valid-sops-stmt t s_1 ∧ valid-sops-stmt t s_2)
|valid-sops-stmt t (While e s) = (valid-sops-expr t e ∧ valid-sops-stmt t s)
|valid-sops-stmt t (SGhost A L R W) = True
|valid-sops-stmt t SFence = True

type-synonym stmt-config = stmt × nat
consts isTrue:: val ⇒ bool

inductive stmt-step:: tmps ⇒ stmt-config ⇒ stmt-config × instrs ⇒ bool
(‹-⊢ - $\rightarrow_s$ -› [60,60,60] 100)
for j
where

AssignAddr:
∀sop. a ≠ Tmp sop ⟹
j⊢ (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 ⊆ dom j ⟹
j⊢ (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 j) (eval-expr t e) (A j) (L j) (R j) (W j)])

| CASAddr:
∀sop. a ≠ Tmp sop ⟹
j⊢ (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:
∀sop. c_e ≠ Tmp sop ⟹
j⊢ (CAS (Tmp (D_a,a)) c_e s_e A L R W, t) $\rightarrow_s$

((CAS (Tmp (D_a,a)) (Tmp (eval-expr t c_e)) s_e A L R W, t + used-tmps c_e), issue-expr t c_e)

| CAS:
$[\![D_a \subseteq$ dom j; $D_c \subseteq$ dom j; eval-expr t s_e = (D,f) $]\!]$
$\Longrightarrow$
j$\vdash$ (CAS (Tmp (D_a,a)) (Tmp (D_c,c)) s_e A L R W, t) $\rightarrow_s$
((Skip, Suc (t + used-tmps s_e)), issue-expr t s_e@
[RMW (a j) (t + used-tmps s_e) (D,f) (λj. the (j (t + used-tmps s_e)) = c j) (λv_1 v_2. v_1)
(A j) (L j) (R j) (W j)])

| Seq:
j$\vdash$ (s_1, t) $\rightarrow_s$ ((s_1', t′), is)
$\Longrightarrow$
j$\vdash$ (Seq s_1 s_2, t) $\rightarrow_s$ ((Seq s_1' s_2, t′),is)

| SeqSkip:
j$\vdash$ (Seq Skip s_2, t) $\rightarrow_s$ ((s_2, t), [])

| Cond:
$\forall$sop. e $\neq$ Tmp sop
$\Longrightarrow$
j$\vdash$ (Cond e s_1 s_2, t) $\rightarrow_s$
((Cond (Tmp (eval-expr t e)) s_1 s_2, t + used-tmps e), issue-expr t e)

| CondTrue:
$[\![$D $\subseteq$ dom j; isTrue (e j)$]\!]$
$\Longrightarrow$
j$\vdash$ (Cond (Tmp (D,e)) s_1 s_2, t) $\rightarrow_s$ ((s_1, t),[])

| CondFalse:
$[\![$D $\subseteq$ dom j; $\neg$ isTrue (e j)$]\!]$
$\Longrightarrow$
j$\vdash$ (Cond (Tmp (D,e)) s_1 s_2, t) $\rightarrow_s$ ((s_2, t),[])

| While:
j$\vdash$ (While e s, t) $\rightarrow_s$
((Cond e (Seq s (While e s)) Skip, t),[])

| SGhost:
j$\vdash$ (SGhost A L R W, t) $\rightarrow_s$ ((Skip, t),[Ghost (A j) (L j) (R j) (W j)])

| SFence:
j$\vdash$ (SFence, t) $\rightarrow_s$ ((Skip, t),[Fence])

inductive-cases stmt-step-cases [cases set]:
j$\vdash$ (Skip, t) $\rightarrow_s$ c
j$\vdash$ (Assign volatile a e A L R W, t) $\rightarrow_s$ c

$j \vdash (\text{CAS a } c_e\ s_e \text{ A L R W, t}) \rightarrow_s c$
$j \vdash (\text{Seq } s_1\ s_2\text{, t}) \rightarrow_s c$
$j \vdash (\text{Cond e } s_1\ s_2\text{, t}) \rightarrow_s c$
$j \vdash (\text{While e s, t}) \rightarrow_s c$
$j \vdash (\text{SGhost A L R W, t}) \rightarrow_s c$
$j \vdash (\text{SFence, t}) \rightarrow_s c$

lemma valid-sops-expr-mono: $\bigwedge t\ t'$. valid-sops-expr t e $\Longrightarrow$ $t \leq t'$ $\Longrightarrow$ valid-sops-expr t' e
⟨*proof*⟩

lemma valid-sops-stmt-mono: $\bigwedge t\ t'$. valid-sops-stmt t s $\Longrightarrow$ $t \leq t'$ $\Longrightarrow$ valid-sops-stmt t' s
⟨*proof*⟩

lemma valid-sops-expr-valid-sop: $\bigwedge t$. valid-sops-expr t e $\Longrightarrow$ valid-sop (eval-expr t e)
⟨*proof*⟩

lemma valid-sops-expr-eval-expr-in-range:
$\bigwedge t$. valid-sops-expr t e $\Longrightarrow$ $\forall t' \in$ fst (eval-expr t e). $t' < t$ + used-tmps e
⟨*proof*⟩

lemma stmt-step-tmps-count-mono:
assumes step: $j \vdash (s,t) \rightarrow_s ((s',t'),is)$
shows $t \leq t'$
⟨*proof*⟩

lemma valid-sops-stmt-invariant:
assumes step: $j \vdash (s,t) \rightarrow_s ((s',t'),is)$
shows valid-sops-stmt t s $\Longrightarrow$ valid-sops-stmt t' s'
⟨*proof*⟩

lemma map-le-restrict-map-eq: $m_1 \subseteq_m m_2 \Longrightarrow D \subseteq \text{dom } m_1 \Longrightarrow m_2 |` D = m_1 |` D$
⟨*proof*⟩

lemma sbh-step-preserves-load-tmps-bound:
assumes step: $(is,\mathcal{O},\mathcal{D},j,sb,\mathcal{S},m) \rightarrow_{sbh} (is',\mathcal{O}',\mathcal{D}',j',sb',\mathcal{S}',m')$
assumes less: $\forall i \in$ load-tmps is. $i < n$
shows $\forall i \in$ load-tmps is'. $i < n$
⟨*proof*⟩

lemma sbh-step-preserves-read-tmps-bound:
assumes step: $(is,j,sb,m,\mathcal{D},\mathcal{O},\mathcal{S}) \rightarrow_{sbh} (is',j',sb',m',\mathcal{D}',\mathcal{O}',\mathcal{S}')$
assumes less-is: $\forall i \in$ load-tmps is. $i < n$
assumes less-sb: $\forall i \in$ read-tmps sb. $i < n$

shows $\forall$ i $\in$ read-tmps sb$'$. i $<$ n
⟨*proof*⟩

lemma sbh-step-preserves-tmps-bound:
assumes step: (is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is$'$,j$'$,sb$'$,m$'$,$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{S}'$)
assumes less-dom: $\forall$ i $\in$ dom j. i $<$ n
assumes less-is: $\forall$ i $\in$ load-tmps is. i $<$ n
shows $\forall$ i $\in$ dom j$'$. i $<$ n
⟨*proof*⟩

lemma flush-step-preserves-read-tmps:
assumes step: (m,sb,$\mathcal{O}$) $\rightarrow_{\mathsf{f}}$ (m$'$,sb$'$,$\mathcal{O}'$)
assumes less-sb: $\forall$ i $\in$ read-tmps sb. i $<$ n
shows $\forall$ i $\in$ read-tmps sb$'$. i $<$ n
⟨*proof*⟩

lemma flush-step-preserves-write-sops:
assumes step: (m,sb,$\mathcal{O}$) $\rightarrow_{\mathsf{f}}$ (m$'$,sb$'$,$\mathcal{O}'$)
assumes less-sb: $\forall$ i$\in\bigcup$(fst ‘ write-sops sb). i $<$ t
shows $\forall$ i$\in\bigcup$(fst ‘ write-sops sb$'$). i $<$ t
⟨*proof*⟩

lemma issue-expr-load-tmps-range$'$:
$\bigwedge$t. load-tmps (issue-expr t e) = {i. t $\leq$ i $\wedge$ i $<$ t + used-tmps e}
⟨*proof*⟩

lemma issue-expr-load-tmps-range:
$\bigwedge$t. $\forall$ i $\in$ load-tmps (issue-expr t e). t $\leq$ i $\wedge$ i $<$ t + (used-tmps e)
⟨*proof*⟩

lemma stmt-step-load-tmps-range$'$:
assumes step: j$\vdash$ (s, t) $\rightarrow_{\mathsf{s}}$ ((s$'$, t$'$),is)
shows load-tmps is = {i. t $\leq$ i $\wedge$ i $<$ t$'$}
⟨*proof*⟩

lemma stmt-step-load-tmps-range:
assumes step: j$\vdash$ (s, t) $\rightarrow_{\mathsf{s}}$ ((s$'$, t$'$),is)
shows $\forall$ i $\in$ load-tmps is. t $\leq$ i $\wedge$ i $<$ t$'$
⟨*proof*⟩

lemma distinct-load-tmps-issue-expr: $\bigwedge$t. distinct-load-tmps (issue-expr t e)
⟨*proof*⟩

lemma max-used-load-tmps: t + used-tmps e $\notin$ load-tmps (issue-expr t e)
⟨*proof*⟩

lemma stmt-step-distinct-load-tmps:
assumes step: $j \vdash (s, t) \rightarrow_{\mathsf{s}} ((s', t'), is)$
shows distinct-load-tmps is
⟨*proof*⟩

lemma store-sops-issue-expr [simp]: $\bigwedge t.$ store-sops (issue-expr t e) = {}
⟨*proof*⟩

lemma stmt-step-data-store-sops-range:
assumes step: $j \vdash (s, t) \rightarrow_{\mathsf{s}} ((s', t'), is)$
assumes valid: valid-sops-stmt t s
shows $\forall (D,f) \in$ store-sops is. $\forall i \in D.\ i < t'$
⟨*proof*⟩

lemma sbh-step-distinct-load-tmps-prog-step:
assumes step: $j \vdash (s,t) \rightarrow_{\mathsf{s}} ((s',t'),is')$
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 $is' \wedge$ (load-tmps $is' \cap$ load-tmps is = {}) $\wedge$
(load-tmps $is' \cap$ read-tmps sb) = {}
⟨*proof*⟩

lemma data-dependency-consistent-instrs-issue-expr:
$\bigwedge t\ T.$ data-dependency-consistent-instrs T (issue-expr t e)
⟨*proof*⟩

lemma dom-eval-expr:
$\bigwedge t.$ ⟦valid-sops-expr t e; $x \in$ fst (eval-expr t e)⟧ $\Longrightarrow x \in \{i.\ i < t\} \cup$ load-tmps (issue-expr t e)
⟨*proof*⟩

lemma Cond-not-s_1: $s_1 \neq$ Cond e s_1 s_2
⟨*proof*⟩

lemma Cond-not-s_2: $s_2 \neq$ Cond e s_1 s_2
⟨*proof*⟩

lemma Seq-not-s_1: $s_1 \neq$ Seq s_1 s_2
⟨*proof*⟩

lemma Seq-not-s_2: $s_2 \neq$ Seq s_1 s_2
⟨*proof*⟩

lemma prog-step-progress:
assumes step: $j \vdash (s,t) \rightarrow_{\mathsf{s}} ((s',t'),is)$
shows $(s',t') \neq (s,t) \vee is \neq []$
⟨*proof*⟩

lemma stmt-step-data-dependency-consistent-instrs:
  assumes step: $j\vdash (s, t) \rightarrow_{s} ((s', t'),is)$
  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: $j\vdash(s,t) \rightarrow_{s} ((s',t'),is')$
  assumes store-sops-le: $\forall i \in \bigcup(fst\ `\ store\text{-}sops\ is).\ i < t$
  assumes write-sops-le: $\forall i \in \bigcup(fst\ `\ write\text{-}sops\ sb).\ i < t$
  assumes valid: valid-sops-stmt t s
  shows data-dependency-consistent-instrs ($\{i.\ i < t\}$) $is' \wedge$
    load-tmps $is' \cap \bigcup(fst\ `\ store\text{-}sops\ is) = \{\} \wedge$
    load-tmps $is' \cap \bigcup(fst\ `\ write\text{-}sops\ sb) = \{\}$
⟨*proof*⟩

lemma sbh-load-tmps-fresh-prog-step:
  assumes step: $j\vdash(s,t) \rightarrow_{s} ((s',t'),is')$
  assumes tmps-le: $\forall i \in dom\ j.\ i < t$
  shows load-tmps $is' \cap dom\ j = \{\}$
⟨*proof*⟩

lemma sbh-valid-sops-prog-step:
  assumes step: $j\vdash(s,t) \rightarrow_{s} ((s',t'),is)$
  assumes valid: valid-sops-stmt t s
  shows $\forall$sop$\in$store-sops is. valid-sop sop
⟨*proof*⟩

primrec prog-configs:: $'a$ memref list $\Rightarrow$ $'a$ set
where
prog-configs [] = {}
|prog-configs (x#xs) = (case x of
      $Prog_{sb}$ p p' is $\Rightarrow$ $\{p,p'\} \cup$ prog-configs xs
     | - $\Rightarrow$ prog-configs xs)

lemma prog-configs-append: $\bigwedge$ys. prog-configs (xs@ys) = prog-configs xs $\cup$ prog-configs ys
  ⟨*proof*⟩

lemma prog-configs-in1: $Prog_{sb}$ p_1 p_2 is $\in$ set xs $\Longrightarrow$ p_1 $\in$ prog-configs xs
  ⟨*proof*⟩

lemma prog-configs-in2: $Prog_{sb}$ p_1 p_2 is $\in$ set xs $\Longrightarrow$ p_2 $\in$ prog-configs xs
  ⟨*proof*⟩

lemma prog-configs-mono: $\bigwedge$ys. set xs $\subseteq$ set ys $\Longrightarrow$ prog-configs xs $\subseteq$ prog-configs ys
  ⟨*proof*⟩

locale separated-tmps =
fixes ts
assumes valid-sops-stmt: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)$]\!]$
  $\Longrightarrow$ valid-sops-stmt t s
assumes valid-sops-stmt-sb: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$); (s$'$,t$'$) $\in$ prog-configs sb$]\!]$
  $\Longrightarrow$ valid-sops-stmt t$'$ s$'$
assumes load-tmps-le: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)$]\!]$
  $\Longrightarrow$ $\forall$i $\in$ load-tmps is. i < t
assumes read-tmps-le: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)$]\!]$
  $\Longrightarrow$ $\forall$i $\in$ read-tmps sb. i < t
assumes store-sops-le: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)$]\!]$
  $\Longrightarrow$ $\forall$i $\in$ $\bigcup$(fst ` store-sops is). i < t
assumes write-sops-le: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)$]\!]$
  $\Longrightarrow$ $\forall$i $\in$ $\bigcup$(fst ` write-sops sb). i < t
assumes tmps-le: $[\![$i < length ts; ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)$]\!]$
  $\Longrightarrow$ dom j $\cup$ load-tmps is = {i. i < t}

lemma (**in** separated-tmps)
  tmps-le$'$:
  assumes i-bound: i < length ts
  assumes ts-i: ts!i = ((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)
  shows $\forall$i $\in$ dom j. i < t
$\langle$*proof*$\rangle$

lemma (**in** separated-tmps) separated-tmps-nth-update:
  $[\![$i < length ts; valid-sops-stmt t s; $\forall$(s$'$,t$'$) $\in$ prog-configs sb. valid-sops-stmt t$'$ s$'$;
  $\forall$i $\in$ load-tmps is. i < t;$\forall$i $\in$ read-tmps sb. i < t;
  $\forall$i $\in$ $\bigcup$(fst ` store-sops is). i < t; $\forall$i $\in$ $\bigcup$(fst ` write-sops sb). i < t; dom j $\cup$ load-tmps is = {i. i < t}$]\!]$
  $\Longrightarrow$
  separated-tmps (ts[i:=((s,t),is,j,sb,$\mathcal{D}$,$\mathcal{O}$)])
  $\langle$*proof*$\rangle$

lemma hd-prog-app-in-first: $\bigwedge$ys. Prog$_{\mathsf{sb}}$ p p$'$ is $\in$ set xs $\Longrightarrow$ hd-prog q (xs @ ys) = hd-prog q xs
  $\langle$*proof*$\rangle$

lemma hd-prog-app-in-eq: $\bigwedge$ys. Prog$_{\mathsf{sb}}$ p p$'$ is $\in$ set xs $\Longrightarrow$ hd-prog q xs = hd-prog x xs
  $\langle$*proof*$\rangle$

lemma hd-prog-app-notin-first: $\bigwedge$ys. $\forall$p p$'$ is. Prog$_{\mathsf{sb}}$ p p$'$ is $\notin$ set xs $\Longrightarrow$ hd-prog q (xs @ ys) = hd-prog q ys
  $\langle$*proof*$\rangle$

lemma union-eq-subsetD: A $\cup$ B = C $\Longrightarrow$ A $\cup$ B $\subseteq$ C $\wedge$ C $\subseteq$ A $\cup$ B
  $\langle$*proof*$\rangle$

lemma prog-step-preserves-separated-tmps:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$)
assumes prog-step: j⊢ p $\rightarrow_{\mathsf{s}}$ (p′, is′)
assumes sep: separated-tmps ts
shows separated-tmps
(ts [i:=(p′,is@is′,j,sb@[Prog$_{\mathsf{sb}}$ p p′ is′],$\mathcal{D}$,$\mathcal{O}$)])
⟨*proof*⟩

lemma flush-step-sb-subset:
assumes step: (m,sb,$\mathcal{O}$) $\rightarrow_{\mathsf{f}}$ (m′, sb′,$\mathcal{O}$′)
shows set sb′ ⊆ set sb
⟨*proof*⟩

lemma flush-step-preserves-separated-tmps:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes flush-step: (m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m′, sb′,$\mathcal{O}$′,$\mathcal{R}$′,$\mathcal{S}$′)
assumes sep: separated-tmps ts
shows separated-tmps (ts [i:=(p,is,j,sb′,$\mathcal{D}$,$\mathcal{O}$′,$\mathcal{R}$′)])
⟨*proof*⟩

lemma sbh-step-preserves-store-sops-bound:
assumes step: (is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is′,j′,sb′,m′,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′,$\mathcal{S}$′)
assumes store-sops-le: ∀ i∈⋃ (fst ‘ store-sops is). i < t
shows ∀ i∈⋃ (fst ‘ store-sops is′). i < t
⟨*proof*⟩

lemma sbh-step-preserves-write-sops-bound:
assumes step: (is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is′,j′,sb′,m′,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′,$\mathcal{S}$′)
assumes store-sops-le: ∀ i∈⋃ (fst ‘ store-sops is). i < t
assumes write-sops-le: ∀ i∈⋃ (fst ‘ write-sops sb). i < t
shows ∀ i∈⋃ (fst ‘ write-sops sb′). i < t
⟨*proof*⟩

lemma sbh-step-prog-configs-eq:
assumes step: (is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is′,j′,sb′,m′,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′,$\mathcal{S}$′)
shows prog-configs sb′ = prog-configs sb
⟨*proof*⟩

lemma sbh-step-preserves-tmps-bound′:
assumes step: (is,j,sb,m,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$ (is′,j′,sb′,m′,$\mathcal{D}$′,$\mathcal{O}$′,$\mathcal{R}$′,$\mathcal{S}$′)
shows dom j ∪ load-tmps is = dom j′ ∪ load-tmps is′
⟨*proof*⟩

lemma sbh-step-preserves-separated-tmps:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes memop-step: (is, j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$

$(is', j', sb', m', \mathcal{D}', \mathcal{O}', \mathcal{R}', \mathcal{S}')$
assumes instr: separated-tmps ts
shows separated-tmps (ts [i:=(p,is',j',sb',$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)])
⟨*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,j,sb::stmt-config store-buffer,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes prog-step: j⊢ p $\rightarrow_{\mathsf{s}}$ (p', is')
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:=(p',is@is',j,sb@[Prog$_{\mathsf{sb}}$ p p' is'],$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)])
⟨*proof*⟩

lemma flush-step-preserves-valid:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb::stmt-config store-buffer,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes flush-step: (m,sb,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{f}}$ (m', sb',$\mathcal{O}'$,$\mathcal{R}'$,$\mathcal{S}'$)
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:=(p,is,j,sb',$\mathcal{D}$,$\mathcal{O}'$,$\mathcal{R}'$)])
⟨*proof*⟩

lemma sbh-step-preserves-valid:
assumes i-bound: i < length ts
assumes ts-i: ts!i = (p,is,j,sb::stmt-config store-buffer,$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)
assumes memop-step: (is, j, sb, m,$\mathcal{D}$, $\mathcal{O}$, $\mathcal{R}$,$\mathcal{S}$) $\rightarrow_{\mathsf{sbh}}$
$(is', j', sb', m', \mathcal{D}', \mathcal{O}', \mathcal{R}', \mathcal{S}')$
assumes valid: valid-pimp ts
shows valid-pimp (ts [i:=(p,is',j',sb',$\mathcal{D}'$,$\mathcal{O}'$,$\mathcal{R}'$)])
⟨*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 λp p' is sb. ()⟨*proof*⟩
interpretation virtual:
computation virtual-memop-step empty-storebuffer-step stmt-step λp p' is sb. ()⟨*proof*⟩
interpretation store-buffer:
computation sb-memop-step store-buffer-step stmt-step λp p' is sb. sb ⟨*proof*⟩

interpretation store-buffer-history:
computation sbh-memop-step flush-step stmt-step λp p′ is sb. sb @ [$\text{Prog}_{\mathsf{sb}}$ p p′ is]⟨*proof*⟩

abbreviation direct-pimp-step::
(stmt-config,unit,bool,owns,rels,shared) global-config $\Rightarrow$ (stmt-config,unit,bool,owns,rels,shared) global-config $\Rightarrow$ bool
(‹- $\Rightarrow_{\mathsf{dp}}$ -› [60,60] 100)
where
c $\Rightarrow_{\mathsf{dp}}$ d $\equiv$ direct.concurrent-step c d

abbreviation direct-pimp-steps::
(stmt-config,unit,bool,owns,rels,shared) global-config $\Rightarrow$ (stmt-config,unit,bool,owns,rels,shared) global-config $\Rightarrow$ bool
(‹- $\Rightarrow_{\mathsf{dp}}{}^*$ -› [60,60] 100)
where
direct-pimp-steps == direct-pimp-step^**
Execution examples**lemma** Assign-Const-ex:
([((Assign True (Tmp ({},λj. a)) (Const c) (λj. A) (λj. L) (λj. R) (λj. W),t),[],j,(),$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)],m,$\mathcal{S}$) $\Rightarrow_{\mathsf{dp}}{}^*$
([((Skip,t),[],j,(),True,$\mathcal{O}$ $\cup$ A $-$ R,Map.empty)],m(a := c),$\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)
⟨*proof*⟩

lemma
([((Assign True (Tmp ({},λj. a)) (Binop (+) (Mem True x) (Mem True y)) (λj. A) (λj. L) (λj. R) (λj. W),t),[],j,(),$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)],m,S)
$\Rightarrow_{\mathsf{dp}}{}^*$
([((Skip,t + 2),[],j(t$\mapsto$m x, t + 1 $\mapsto$m y),(),True,$\mathcal{O}$ $\cup$ A $-$ R,Map.empty)],m(a := m x + m y),S $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)
⟨*proof*⟩

lemma
assumes isTrue: isTrue c
shows
([((Cond (Const c) (Assign True (Tmp ({},λj. a)) (Const c) (λj. A) (λj. L) (λj. R) (λj. W)) Skip,t) ,[],j,(),$\mathcal{D}$,$\mathcal{O}$,$\mathcal{R}$)],m,$\mathcal{S}$) $\Rightarrow_{\mathsf{dp}}{}^*$
([((Skip,t),[],j,(),True,$\mathcal{O}$ $\cup$ A $-$ R,Map.empty)],m(a := c),$\mathcal{S}$ $\oplus_{\mathsf{W}}$ R $\ominus_{\mathsf{A}}$ L)
⟨*proof*⟩

end

References

1. Advanced Micro Devices (AMD), Inc. *AMD64 Architecture Programmer's Manual: Volumes 1–3*. September 2007. rev. 3.14.
2. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
3. David Aspinall and Jaroslav Sevcík. Formalising Java's data race free guarantee. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732, pages 22–37, 2007.
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,*

Torino, Italy, April 30 – May 4, 2003, Revised Selected Papers, volume 3085, pages 34–50. Springer, 2003.

5. Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11–12, 2006, Proceedings*, volume 4108, pages 31–43. Springer, 2006.
6. Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 107–120, Berlin, Heidelberg, 2008. Springer-Verlag.
7. Geng Chen, Ernie Cohen, and Mikhail Kovalev. Store buffer reduction with MMUs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, pages 117–132, Cham, 2014. Springer International Publishing.
8. Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In Matt Kaufmann, Lawrence Paulson, and Michael Norrish, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lecture Notes in Computer Science*, Edinburgh, UK, 2010. Springer.
9. Intel. Intel 64 architecture memory ordering white paper. SKU 318147-001, 2007.
10. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volumes 1–3b*. 2009. rev. 29.
11. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180, pages 180–192, 1996.
12. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.
13. Jonas Oberhauser. A simpler reduction theorem for x86-tso. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Verified Software: Theories, Tools, and Experiments*, pages 142–164, Cham, 2016. Springer International Publishing.
14. Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. Technical report, University of Cambridge, 2009.
15. Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, 2009.
16. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828. Springer, 1994.
17. Tom Ridge. Operational reasoning for concurrent Caml programs and weak memory models. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732, pages 278–293, 2007.
18. Jaroslav Sevcík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP*, volume 5142, pages 27–51, 2008.