A Better Reduction Theorem for Store Buffers

Ernie Cohen[1] and Norbert Schirmer[2],[*]

[1] Microsoft Corp., Redmond, WA, USA
[2] German Research Center for Artificial Intelligence (DFKI) Saarbrücken, Germany
`ernie.cohen@microsoft.com`, `norbert.schirmer@dfki.de`

Abstract. When verifying a concurrent program, it is usual to assume that memory is sequentially consistent. However, most modern multi-processors 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.

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 P0 and P1, where x and y are shared memory variables (initially 0) and r0 and r1 are registers:

```
P0              P1

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]

---

[3] Before 2008, Intel [7] 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 [8]. According to Owens et al. [11] AMD is also planning a similar adaptation of their manuals.

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 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

---

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

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 metatheory.

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, whether the thread has performed a write since the last flush, and which owned addresses were acquired 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 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.

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. Finally we conclude in Section 5.

2 Preliminaries

The formalization presented in this papaer is mechanized and checked within the generic interactive theorem prover *Isabelle* [12]. 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* [10].

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

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

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

Isabelle/HOL provides a library of standard types like Booleans, natural numbers, integers, total functions, pairs, lists, and sets. Moreover, there are packages to define new data types and records. Isabelle allows polymorphic types, e.g., $'a$ list is the list type with type variable $'a$. In HOL all functions are total, e.g., nat $\Rightarrow$ nat is a total function on natural numbers. A function update is $f(y := v) \equiv \lambda x.\ \texttt{if}\ x = y\ \texttt{then}\ v\ \texttt{else}\ f\ x$. To formalize partial functions the type $'a$ option is used. It is a data type with two constructors, one to inject

5

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 the $\lfloor x \rfloor = x$. Since HOL is a total logic the term the $\bot$ is still a well-defined yet un(der)specified value. Partial functions are usually represented by the type $'a \Rightarrow 'b$ option, abbreviated as $'a \rightharpoonup 'b$. They are commonly used as *maps*. We denote the domain of map m by dom m. A map update is written as $m(a \mapsto v)$. 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 $x \cdot xs$ the element x is 'consed' to the list xs. With xs @ ys list ys is appended to list xs. With the term map f xs the function f is applied to all elements in xs. The length of a list is $|xs|$, the n-th element of a list can be selected with $xs_{[n]}$ and can be updated via $xs[n := v]$. With dropWhile P xs the prefix for which all elements satisfy predicate P are dropped from list xs.

Sets come along with the standard operations like union, i.e., $A \cup B$, 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 the 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 special ghost operations. 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. 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
owned	vR, vW, R	unreachable	vR, vW, R, W
owned by other	vR	unreachable	
unowned	vR, vW	vR, R	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 the computation of the programming language memory instructions (read / write) are issued 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 then describe the virtual machine and in Section 4.2 the store buffer machine. Section 4.3 gives some details of our coupling relation which is used for the simulation proof presented in Section 4.4. Finally, in Section 4.5 we illustrate the integration of a programming language on top of the memory system, by presenting PIMP, a concurrent variant of a WHILE language.

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 acquire addresses. The possible status changes of an address due to these ownership transfer operations are depicted in Figure 1. A

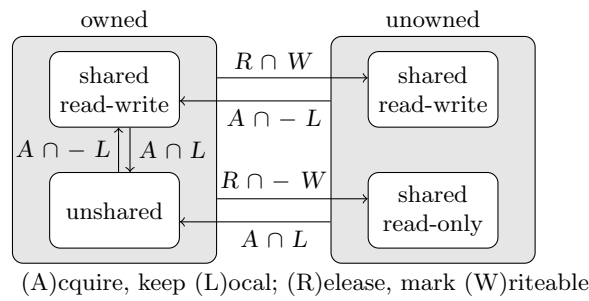(A)cquire, keep (L)ocal; (R)elease, mark (W)riteable

Fig. 1: Ownership transfer

memory instruction is a datatype with the following constructors:

- READ *volatile a t* for reading from address *a* to temporary *t*, where the Boolean *volatile* determines whether the access is volatile or not.
- WRITE *volatile a sop A L R W* to write the result of evaluating the store operation *sop* at address *a*. A store operation is a pair (D, f), with the domain *D* and the function *f*. The function *f* takes temporaries ϑ 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:

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

  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* to acquire ownership on addresses.

The configuration of a single thread is a tuple $(p, \mathit{is}, \vartheta, \mathit{sb}, \mathcal{D}, \mathcal{O})$ consisting of the program state *p*, a memory instruction list *is*, the map of temporaries ϑ, the store buffer *sb*, a dirty flag $\mathcal{D}$ indicating whether there may be an outstanding volatile write in the store buffer and finally 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 type of the program state *p* and the store buffer *sb* is free. For example we later instantiate the store buffer with the union type in case of the virtual machine or with a list of store buffer instructions in case of the machine with store buffer.

A global configuration $(\mathit{ts}, \mathcal{S}, m)$ consists of a list of thread configurations *ts*, a Boolean map of shared addresses $\mathcal{S}$ (indicating write permission) and the memory *m*, which is a function from addresses to values. Addresses in the domain of mapping $\mathcal{S}$ are considered shared and the set of read-only addresses is obtained from *S* by: read-only $\mathcal{S} \equiv \{a.\ \mathcal{S}\ a = \lfloor \mathsf{False} \rfloor\}$

We describe the computation of the global system by the non-deterministic transition relation $(\mathit{ts}, \mathcal{S}, m) \Rightarrow (\mathit{ts'}, \mathcal{S'}, m')$ defined in Figure 2. A transition selects a thread $\mathit{ts}_{[i]} = (p, \mathit{is}, \vartheta, \mathit{sb}, \mathcal{D}, \mathcal{O})$ and either the 'program' the 'memory' or the 'store buffer' makes a step. These three sub-relations are parameters to the global transition relation. The ownership information stored in the ghost components $\mathcal{D}$, $\mathcal{O}$, and $\mathcal{S}$ is sometimes grouped as a single component $\mathcal{G}$ in the transition rules for succinct presentation.

A program step $\vartheta \vdash p \rightarrow_{\mathsf{p}} (p', \mathit{is'})$ takes the temporaries ϑ and the current program state *p* and makes a step by returning a new program state p' and an

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

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

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

Fig. 2: Global transitions

instruction list *is'* which is appended to the remaining instructions. With the functional parameter *record* we are able to maintain bookkeeping information about the program step within the store buffer. It takes the program states p and p', the issued instructions *is'* and the store buffer *sb* as a parameter. This is a technical device in our proof which allows us to remember program steps of the store buffer machine that are still pending in the virtual machine.

A memory step $(is,\ \vartheta,\ sb,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S}) \rightarrow_{\mathsf{m}} (is',\ \vartheta',\ sb',\ m',\ \mathcal{D}',\ \mathcal{O}',\ \mathcal{S}')$ of a machine with store buffer may only fill its store buffer.

In a store buffer step $(m,\ sb,\ \mathcal{O},\ \mathcal{S}) \rightarrow_{\mathsf{sb}} (m',\ sb',\ \mathcal{O}',\ \mathcal{S}')$ the store buffer may release outstanding instructions to the memory.

4.1 Virtual machine

The virtual machine is a sequentially consistent machine without store buffers. The transition rules for its memory system are defined in Figure 3. The store buffer, which is irrelevant in this transition system is referenced by x. We instantiate the global transition system with these rules for the memory system, and the identity relation for store buffer steps, the program steps are still a parameter. We refer to a transition by $(ts,\ \mathcal{S},\ m) \overset{\mathsf{v}}{\Rightarrow} (ts',\ \mathcal{S}',\ m')$.

In addition to the transition rules for the virtual machine we introduce the *safety* judgment $\mathcal{O}s, i \vdash (is,\ \vartheta,\ x,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \sqrt{}$ in Figure 4, where $\mathcal{O}s$ is the list of ownership sets obtained from the thread list *ts* and i is the thread index. Safety of all reachable states of the virtual machine ensures that the access policy 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 Figures 3 and 4 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

$$\overline{(\text{READ } volatile \ a \ t \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{G}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta(t \mapsto m \ a), \ x, \ m, \ \mathcal{G})}$$

$$\overline{(\text{WRITE } \mathsf{False} \ a \ (D, \ f) \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{G}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta, \ x, \ m(a := f \ \vartheta), \ \mathcal{G})}$$

$$\frac{\mathcal{G} = (\mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{True}, \ \mathcal{O} \cup A - R, \ \mathcal{S} \oplus_W R \ominus_A L)}{(\text{WRITE } \mathsf{True} \ a \ (D, \ f) \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{G}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta, \ x, \ m(a := f \ \vartheta), \ \mathcal{G}')}$$

$$\frac{\neg \ cond \ (\vartheta(t \mapsto m \ a)) \qquad \mathcal{G} = (\mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{False}, \ \mathcal{O}, \ \mathcal{S})}{(\text{RMW } a \ t \ (D, \ f) \ cond \ ret \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{G}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta(t \mapsto m \ a), \ x, \ m, \ \mathcal{G}')}$$

$$\frac{\begin{array}{c} cond \ (\vartheta(t \mapsto m \ a)) \\ \vartheta' = \vartheta(t \mapsto ret \ (m \ a) \ (f \ (\vartheta(t \mapsto m \ a)))) \qquad m' = m(a := f \ (\vartheta(t \mapsto m \ a))) \\ \mathcal{G} = (\mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{False}, \ \mathcal{O} \cup A - R, \ \mathcal{S} \oplus_W R \ominus_A L) \end{array}}{(\text{RMW } a \ t \ (D, \ f) \ cond \ ret \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{G}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta', \ x, \ m', \ \mathcal{G}')}$$

$$\overline{(\text{FENCE } \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta, \ x, \ m, \ \mathsf{False}, \ \mathcal{O}, \ \mathcal{S})}$$

$$\overline{(\text{GHOST } A \ L \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \xrightarrow{\text{v}}_{\text{m}} (is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O} \cup A, \ \mathcal{S} \ominus_A L)}$$

Fig. 3: Memory transitions of the virtual machine

$$\frac{a \in \mathcal{O} \vee a \in \mathsf{read\text{-}only} \ \mathcal{S} \vee volatile \wedge a \in \mathsf{dom} \ \mathcal{S} \qquad volatile \longrightarrow \neg \ \mathcal{D}}{\mathcal{O}s,i \vdash (\text{READ } volatile \ a \ t \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \ \sqrt{}}$$

$$\frac{a \in \mathcal{O} \qquad a \notin \mathsf{dom} \ \mathcal{S}}{\mathcal{O}s,i \vdash (\text{WRITE } \mathsf{False} \ a \ (D, \ f) \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \ \sqrt{}}$$

$$\frac{\begin{array}{c} \forall j < |\mathcal{O}s|. \ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \\ a \notin \mathsf{read\text{-}only} \ \mathcal{S} \qquad \forall j < |\mathcal{O}s|. \ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset \\ A \subseteq \mathcal{O} \cup \mathsf{dom} \ \mathcal{S} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset \end{array}}{\mathcal{O}s,i \vdash (\text{WRITE } \mathsf{True} \ a \ (D, \ f) \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \ \sqrt{}}$$

$$\frac{\neg \ cond \ (\vartheta(t \mapsto m \ a)) \qquad a \in \mathsf{dom} \ \mathcal{S} \cup \mathcal{O}}{\mathcal{O}s,i \vdash (\text{RMW } a \ t \ (D, \ f) \ cond \ ret \ A \ L \ R \ W \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \ \sqrt{}}$$

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

$$\overline{\mathcal{O}s,i \vdash (\text{FENCE } \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \ \sqrt{}}$$

$$\frac{A \subseteq \mathsf{dom} \ \mathcal{S} \cup \mathcal{O} \qquad L \subseteq A \qquad \forall j < |\mathcal{O}s|. \ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset}{\mathcal{O}s,i \vdash (\text{GHOST } A \ L \cdot is, \ \vartheta, \ x, \ m, \ \mathcal{D}, \ \mathcal{O}, \ \mathcal{S}) \ \sqrt{}}$$

Fig. 4: Safe configurations of a virtual machine

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 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:

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

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

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 allows to acquire 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. Releasing ownership can always be delayed to the next volatile (or interlocked) write instruction, since only with the write another thread can gain information about released addresses. In the simulation proof we build on the fact that all pending ghost operations in the store buffer until the first volatile write may only acquire addresses to the ownership of the thread.

4.2   Store buffer machine

The store buffer machine extends the virtual machine by maintaining a list of outstanding memory writes. Write instructions are appended to the store buffer and read instructions are satisfied from the store buffer if possible. To support our coupling relation between a configuration of the store buffer machine and a configuration of the virtual machine, we also maintain additional bookkeeping information inside the store buffer. For every write we keep the volatile flag and the store operation. Moreover we record read, program and ghost steps.

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

- READ$_{\mathsf{sb}}$ *volatile a t v*, recording a corresponding read from address a which loaded the value v to temporary t, or a
- WRITE$_{\mathsf{sb}}$ *volatile a sop v* for an outstanding write, where operation *sop* evaluated to value v, or of the form
- PROG$_{\mathsf{sb}}$ p p' *is'*, recording a program transition from p to p' which issued instructions *is'*, or of the form
- GHOST$_{\mathsf{sb}}$ A L, recording a corresponding ghost operation to acquire addresses A and keep addresses L local.

As defined in Figure 5 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 information is analogous to the corresponding operations in the virtual machine. The transitions defined in Figure 6 are straightforward

$$\frac{}{(m, \text{WRITE}_{\mathsf{sb}} \; \mathsf{False} \; a \; sop \; v \; A \; L \; R \; W \cdot sb, \mathcal{O}, \mathcal{S}) \rightarrow_{\mathsf{sb}} (m(a := v), sb, \mathcal{O}, \mathcal{S})}$$

$$\frac{\mathcal{O}' = \mathcal{O} \cup A - R \qquad \mathcal{S}' = \mathcal{S} \oplus_W R \ominus_A L}{(m, \text{WRITE}_{\mathsf{sb}} \; \mathsf{True} \; a \; sop \; v \; A \; L \; R \; W \cdot sb, \mathcal{O}, \mathcal{S}) \rightarrow_{\mathsf{sb}} (m(a := v), sb, \mathcal{O}', \mathcal{S}')}$$

$$\frac{}{(m, \text{READ}_{\mathsf{sb}} \; volatile \; a \; t \; v \cdot sb, \mathcal{O}, \mathcal{S}) \rightarrow_{\mathsf{sb}} (m, sb, \mathcal{O}, \mathcal{S})}$$

$$\frac{}{(m, \text{PROG}_{\mathsf{sb}} \; p \; p' \; is \cdot sb, \mathcal{O}, \mathcal{S}) \rightarrow_{\mathsf{sb}} (m, sb, \mathcal{O}, \mathcal{S})}$$

$$\frac{}{(m, \text{GHOST}_{\mathsf{sb}} \; A \; L \cdot sb, \mathcal{O}, \mathcal{S}) \rightarrow_{\mathsf{sb}} (m, sb, \mathcal{O} \cup A, \mathcal{S} \ominus_A L)}$$

Fig. 5: Store buffer transitions

extensions of the transitions of the virtual machine. 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. This also includes their effect on the ownership information. 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. We instantiate the global transition system with the rules of Figures 5 and 6. The program transitions are still a parameter. We refer to a transition by $(ts, \mathcal{S}, m) \overset{\mathsf{sb}}{\Rightarrow} (ts', \mathcal{S}', m')$.

13

$$v = (\text{case buffered-val } sb \; a \; \text{of} \perp \Rightarrow m \; a \mid \lfloor v' \rfloor \Rightarrow v')$$
$$\frac{sb' = sb \; @ \; [\text{READ}_{\sf sb} \; volatile \; a \; t \; v]}{(\text{READ} \; volatile \; a \; t \cdot is, \; \vartheta, \; sb, \; m, \; \mathcal{G}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta(t \mapsto v), \; sb', \; m, \; \mathcal{G})}$$

$$\frac{sb' = sb \; @ \; [\text{WRITE}_{\sf sb} \; {\sf False} \; a \; (D, f) \; (f \; \vartheta) \; A \; L \; R \; W]}{(\text{WRITE} \; {\sf False} \; a \; (D, f) \; A \; L \; R \; W \cdot is, \; \vartheta, \; sb, \; m, \; \mathcal{G}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta, \; sb', \; m, \; \mathcal{G})}$$

$$\frac{sb' = sb \; @ \; [\text{WRITE}_{\sf sb} \; {\sf True} \; a \; (D, f) \; (f \; \vartheta) \; A \; L \; R \; W] \qquad \mathcal{G} = (\mathcal{D}, \mathcal{O}, \mathcal{S}) \qquad \mathcal{G}' = ({\sf True}, \mathcal{O}, \mathcal{S})}{(\text{WRITE} \; {\sf True} \; a \; (D, f) \; A \; L \; R \; W \cdot is, \; \vartheta, \; sb, \; m, \; \mathcal{G}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta, \; sb', \; m, \; \mathcal{G}')}$$

$$\frac{\neg \; cond \; (\vartheta(t \mapsto m \; a)) \qquad \mathcal{G} = (\mathcal{D}, \mathcal{O}, \mathcal{S}) \qquad \mathcal{G}' = ({\sf False}, \mathcal{O}, \mathcal{S})}{(\text{RMW} \; a \; t \; (D, f) \; cond \; ret \; A \; L \; R \; W \cdot is, \; \vartheta, \; [], \; m, \; \mathcal{G}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta(t \mapsto m \; a), \; [], \; m, \; \mathcal{G}')}$$

$$cond \; (\vartheta(t \mapsto m \; a))$$
$$\frac{\vartheta' = \vartheta(t \mapsto ret \; (m \; a) \; (f \; (\vartheta(t \mapsto m \; a)))) \qquad m' = m(a := f \; (\vartheta(t \mapsto m \; a))) \qquad \mathcal{G} = (\mathcal{D}, \mathcal{O}, \mathcal{S}) \qquad \mathcal{G}' = ({\sf False}, \mathcal{O} \cup A - R, \mathcal{S} \oplus_W R \ominus_A L)}{(\text{RMW} \; a \; t \; (D, f) \; cond \; ret \; A \; L \; R \; W \cdot is, \; \vartheta, \; [], \; m, \; \mathcal{G}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta', \; [], \; m', \; \mathcal{G}')}$$

$$\frac{}{(\text{FENCE} \cdot is, \; \vartheta, \; [], \; m, \; \mathcal{D}, \mathcal{O}, \mathcal{S}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta, \; [], \; m, \; {\sf False}, \mathcal{O}, \mathcal{S})}$$

$$\frac{}{(\text{GHOST} \; A \; L \cdot is, \; \vartheta, \; sb, \; m, \; \mathcal{D}, \mathcal{O}, \mathcal{S}) \xrightarrow{\sf sb}_{\sf m} (is, \; \vartheta, \; sb \; @ \; [\text{GHOST}_{\sf sb} \; A \; L], \; m, \; \mathcal{D}, \mathcal{O}, \mathcal{S})}$$

Fig. 6: Memory transitions of store buffer machine

4.3 Coupling relation

In this section we establish the coupling relation between a configuration of a machine with store buffer and the virtual machine without store buffer. 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. The same is true for reads which get recorded in the store buffer. 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 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 access policy 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. Altogether this suggests the following coupling relation: the memory of the virtual machine is obtained from the memory of the store buffer machine, by flushing every store buffer until we reach a volatile write. The remaining store buffer entries are suspended as instructions. The suspended reads are not yet visible in the temporaries of the virtual machine. Similar the ownership effects of the suspended ghost operations is not yet visible in the virtual machine.

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

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

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 ϑ'. Similar, the suspended updates to the ownership sets and dirty flag are not yet recorded in $\mathcal{O}'$, and $\mathcal{D}'$.

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

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:

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

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 $(ts_{\mathsf{sb}},\ \mathcal{S}_{\mathsf{sb}},\ m_{\mathsf{sb}}) \sim (ts,\ \mathcal{S},\ m)$, defined in Figure 7. We denote the already simulated (flushed) store buffer en-

$$
\begin{array}{c}
m = \mathsf{flush\text{-}all\text{-}until\text{-}volatile\text{-}write}\ ts_{\mathsf{sb}}\ m_{\mathsf{sb}} \\
\mathcal{S} = \mathsf{share\text{-}all\text{-}until\text{-}volatile\text{-}write}\ ts_{\mathsf{sb}}\ \mathcal{S}_{\mathsf{sb}} \qquad |ts_{\mathsf{sb}}| = |ts| \\
\forall\, i < |ts_{\mathsf{sb}}|. \\
\quad \mathtt{let}\ (p,\ is_{\mathsf{sb}},\ \theta,\ sb,\ \mathcal{D}_{\mathsf{sb}},\ \mathcal{O}) = ts_{\mathsf{sb}[i]}; \\
\qquad\quad flushs = \mathsf{takeWhile\ not\text{-}volatile\text{-}write}\ sb; \\
\qquad\quad suspends = \mathsf{dropWhile\ not\text{-}volatile\text{-}write}\ sb \\
\quad \mathtt{in}\ \exists\, is\ \mathcal{D}.\ \mathsf{instrs}\ suspends \mathbin{@} is_{\mathsf{sb}} = is \mathbin{@} \mathsf{prog\text{-}instrs}\ suspends\ \wedge \\
\qquad\quad \mathcal{D}_{\mathsf{sb}} = (\mathcal{D} \vee \mathsf{refs\ volatile\text{-}Write}\ sb \neq \emptyset)\ \wedge \\
\qquad\quad ts_{[i]} = \\
\qquad\quad (\mathsf{hd\text{-}prog}\ p\ suspends,\ is,\ \theta{\upharpoonright}_{(-\ \mathsf{read\text{-}tmps}\ suspends)},\ (),\ \mathcal{D}, \\
\qquad\quad\ \mathsf{acquire}\ flushs\ \mathcal{O}) \\
\hline
(ts_{\mathsf{sb}},\ \mathcal{S}_{\mathsf{sb}},\ m_{\mathsf{sb}}) \sim (ts,\ \mathcal{S},\ m)
\end{array}
$$

Fig. 7: Coupling relation

tries by *flushs* and the suspended ones by *suspends*. The function instrs converts them back to instructions, which are a prefix of the instructions of the virtual machine. We collect the additional instructions which were issued by program instructions but still recorded in the remainder of the store buffer with function 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 instrs *suspends* @ is_{sb} = *is* @ 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 ϑ. The memory is obtained by flushing all store buffers until the first volatile write is hit, excluding it. Thereby only non-volatile writes are flushed, which are all thread local, and hence could be flushed in any order with the same result on the memory. Function flush-all-until-volatile-write flushes them in order of appearance. Similarly the sharing map of the virtual machine is obtained by flushing all store buffers until the first volatile write via the function share-all-until-volatile-write. For the local ownership set $\mathcal{O}$ the auxiliary function acquire calculates the outstanding effect of the already simulated parts of the store buffer.

One may think of simplifying the coupling relation by avoiding flushing altogether and just suspending the whole store buffer. 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, and we could miss updates made by other threads to this address.

4.4 Simulation

Theorem 1 is our core simulation theorem. Provided that all reachable states of the virtual machine are safe, a step of the store buffer machine 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 1 (Simulation).

$(ts_{sb}, \mathcal{S}_{sb}, m_{sb}) \overset{sb}{\Rightarrow} (ts_{sb}{}', \mathcal{S}_{sb}{}', m_{sb}{}') \wedge (ts_{sb}, \mathcal{S}_{sb}, m_{sb}) \sim (ts, \mathcal{S}, m) \wedge$
$\quad$ safe-reach $(ts, \mathcal{S}, m) \wedge$ invariant $ts_{sb}\ \mathcal{S}_{sb}\ m_{sb} \longrightarrow$
$\quad$ invariant $ts_{sb}{}'\ \mathcal{S}_{sb}{}'\ m_{sb}{}' \wedge$
$\quad (\exists\, ts'\ \mathcal{S}'\ m'.\ (ts, \mathcal{S}, m) \overset{v}{\Rightarrow}{}^* (ts', \mathcal{S}', m') \wedge (ts_{sb}{}', \mathcal{S}_{sb}{}', m_{sb}{}') \sim (ts', \mathcal{S}', m'))$

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

invariant $ts_{sb}\ S\ m_{sb} \equiv$ ownership-inv $S\ ts_{sb} \wedge$ sharing-inv $S\ ts_{sb} \wedge$
$\quad$ temporaries-inv $ts_{sb} \wedge$ data-dependency-inv $ts_{sb} \wedge$ history-inv $ts_{sb}\ m_{sb} \wedge$
$\quad$ flush-inv $ts_{sb} \wedge$ valid ts_{sb}

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.

17

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 ϑ.

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 WRITE *volatile a* (D, f) A L R W $\cdot$ *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 flushed in the virtual machine and thus cause no trouble. (iii) For every read the recorded value coincides with the corresponding value in the temporaries. (iv) For every WRITE$_\mathsf{sb}$ *volatile a* (D, f) v A L R W the recorded value v coincides with f ϑ, and domain D is subset of dom ϑ and is distinct from the following read temporaries. Note that the consistency of the ownership annotations is already covered by the aforementioned invariants. (v) For every suffix in the store buffer of the form PROG$_\mathsf{sb}$ p_1 p_2 *is'* $\cdot$ *sb'*, either $p_1 = p$ in case there is no preceding program node in the buffer or it corresponds to the last program state recorded there. Moreover, the program transition $\vartheta|_{(-\ \mathsf{read\text{-}tmps}\ sb')} \vdash p_1 \rightarrow_\mathsf{p} (p_2,\ is')$ is possible, i.e., it was possible to execute the program transition at that point. (vi) The program configuration p coincides with the last program configuration recorded in the store buffer. (vii) As the instructions from a program step are at the one hand appended to the instruction list and on the other hand recorded in the store buffer, we have for every suffix sb' of the store buffer: $\exists$ *is'*. instrs sb' @ *is* = *is'* @ prog-instrs sb', i.e., the remaining instructions *is* correspond to a suffix of the recorded instructions prog-instrs sb'.

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

Program step. The program-transitions are still a parameter of our model. In order to make the proof work, we have to assume some of the invariants also for the program steps. We allow the program-transitions to employ further invariants on the configurations, these are modeled by the parameter *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 ϑ 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. We do not go into details but rather 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. 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 flushed 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 6. 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 5. 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.

A common argument in various places in the proof is to rule out potential races by constructing calculations of the virtual machine that lead to an unsafe state and are thus unreachable in a safe execution. Here we make use of the fact, that the ghost operations in the prefixes of the store buffers that are already simulated in the virtual machine may only acquire new addresses to the ownership of a thread but not realease addresses. From the viewpoint of other threads this may only lead to a more restrictive configuration but never to more liberal

one, making the construction of an unsafe execution of the virtual machine possible without referring to an older state as the current state of the store buffer machine.

4.5 PIMP

PIMP is a parallel version of IMP [9], a canonical WHILE-language.

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

A statement s is either (i) SKIP, the empty statement, (ii) ASSIGN *volatile* a e A L R W, a (volatile) assignment of expression e to address expression a, (iii) CAS a c_e s_e A L R W, atomic compare and swap at address expression a with compare expression c_e and swap expression s_e, (iv) SEQ s_1 s_2, sequential composition, (v) COND e s_1 s_2, the if-then-else statement, (vi) WHILE e s, the loop statement with condition e, (vii) SGHOST, and SFENCE as stubs for the corresponding memory instructions.

The key idea of the semantics is the following: expressions are evaluated by issuing instructions to the memory system, then the program waits until the memory system has made all necessary results available in the temporaries, which allows the program to make another step. Figure 8 defines expression evaluation. The function used-tmps e calculates the number of temporaries that

$$
\begin{aligned}
\text{issue-expr } t \ (\text{CONST } v) &= [] \\
\text{issue-expr } t \ (\text{MEM } volatile \ a) &= [\text{READ } volatile \ a \ t] \\
\text{issue-expr } t \ (\text{TMP } (D, f)) &= [] \\
\text{issue-expr } t \ (\text{UNOP } f \ e) &= \text{issue-expr } t \ e \\
\text{issue-expr } t \ (\text{BINOP } f \ e_1 \ e_2) &= \text{issue-expr } t \ e_1 \ @ \ \text{issue-expr } (t + \text{used-tmps } e_1) \ e_2 \\
\\
\text{eval-expr } t \ (\text{CONST } v) &= (\emptyset, \lambda\theta. \ v) \\
\text{eval-expr } t \ (\text{MEM } volatile \ a) &= (\{t\}, \lambda\theta. \ \text{the } (\theta \ t)) \\
\text{eval-expr } t \ (\text{TMP } (D, f)) &= (D, f) \\
\text{eval-expr } t \ (\text{UNOP } f \ e) &= \text{let } (D, f_e) = \text{eval-expr } t \ e \ \text{in } (D, \lambda\theta. \ f \ (f_e \ \theta)) \\
\text{eval-expr } t \ (\text{BINOP } f \ e_1 \ e_2) &= \text{let } (D_1, f_1) = \text{eval-expr } t \ e_1; \\
&\qquad \quad (D_2, f_2) = \text{eval-expr } (t + \text{used-tmps } e_1) \ e_2 \\
&\quad \text{in } (D_1 \cup D_2, \lambda\theta. \ f \ (f_1 \ \theta) \ (f_2 \ \theta))
\end{aligned}
$$

Fig. 8: Expression evaluation

are necessary to evaluate expression e, where every MEM expression accounts to one temporary. With issue-expr t e we obtain the instruction list for expression e starting at temporary t, whereas eval-expr t e constructs the operation as a pair of the domain and a function on the temporaries.

The program transitions are defined in Figure 9. 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 by stepwise unfolding. Ghost and fence statements are just propagated to the memory system. To instantiate Theorem 1 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 ϑ be the valuation of temporaries in the current configuration, for every thread configuration $ts_{sb[i]} = ((s, t), is, \vartheta, sb, \mathcal{D}, \mathcal{O})$ where $i < |ts_{sb}|$ we require: (i) The domain of all intermediate TMP (D, f) expressions in statement s is below counter t. (ii) All temporaries in the memory system including the store buffer are below counter t. (iii) All temporaries less than counter t are either already defined in the temporaries ϑ 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.

5 Conclusion

We have presented a practical and flexible concurrent programming discipline that ensures sequential consistency on TSO machines, such as present in current 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,

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

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

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

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

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

$$\dfrac{\vartheta \vdash (s_1,\ t) \rightarrow_{\mathsf{p}} ((s_1',\ t'),\ is)}{\vartheta \vdash (\textsc{Seq}\ s_1\ s_2,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Seq}\ s_1'\ s_2,\ t'),\ is)}$$

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

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

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

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

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

$$\overline{\vartheta \vdash (\text{SGhost}\ A\ L,\ t) \rightarrow_{\mathsf{p}} ((\textsc{Skip},\ t),\ [\textsc{Ghost}\ (A\ \vartheta)\ (L\ \vartheta)])}$$

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

Fig. 9: Program transitions

namely the adherence to the access and flushing policy which can be discharged within any program logic.

Related work. A categorization of various weak memory models is presented in [2]. It is compatible with the recent revisions of the Intel manuals [8] and the revised x86 model presented in [11]. 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, 14] or the x86 memory model [11]. Ridge [13] 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.

Future work. We currently have an asymmetry in the ghost operations for ownership transfer: whereas we have a 'free flowing' ghost operation to acquire an address which can appear anywhere, releases are delayed to the next (volatile or interlocked) write operation. As sketched in Section 4.4 delaying releases is motivated by our simulation proof. To rule out certain races we argue that the state of the virtual machine is at most more restrictive as the state of the store buffer machine. As the virtual state is obtained by flushing all store buffers until the first volatile write is hit (excluding it) a release in that flushed section would violate this invariant. However, we believe this does not restrict expressibility as the only way for other threads to gain knowledge about the released address is via the next volatile write operation of the thread. Formally we want to liberate the points where an ownership release can happen by introducing free flowing releases. The informal key argument is that an unsafe execution with delayed releases implies an unsafe execution with free flowing releases. Between the release point in the free flowing execution and the delayed release point (at the next volatile write) a thread only executes commuting operations with respect to bad races. Such a race at the delayed point already justifies a race at the release point in the free flowing execution.

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.

Finally, we should note that there are important programs that, in the presence of store buffers, are correct but not sequentially consistent. A typical example is the following simplified form of barrier synchronization: each processor has a flag that it writes 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. One possibility is to give a more general reduction theorem that allows each processor to always treat store buffers of other processors as empty, and its own store buffer as empty except for brief periods of time.

Acknowledgements

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. Intel. Intel 64 architecture memory ordering white paper. SKU 318147-001, 2007.

8. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volumes 1–3b.* 2009. rev. 29.

9. 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.

10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002.

11. 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.

12. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828. Springer, 1994.

13. 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.

14. 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.