Verifying TSO programs

Bart Jacobs

Report CW 660, May 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verifying TSO programs

Bart Jacobs

Report CW 660, May 2014

Department of Computer Science, KU Leuven

Abstract

TSO (Total Store Order) is the memory consistency model implemented by the x86 and x64 architectures. While for data-race-free programs the stronger SC (Sequential Consistency) memory consistency model can be assumed, some programs escape from the SC constraints for performance reasons. In this document we propose an approach for verifying programs under the TSO memory consistency model.

Verifying TSO programs

Bart Jacobs
iMinds-DistriNet, Dept. Comp. Sci., KU Leuven, Belgium
`bart.jacobs@cs.kuleuven.be`

Abstract

TSO (Total Store Order) is the memory consistency model implemented by the x86 and x64 architectures. While for data-race-free programs the stronger SC (Sequential Consistency) memory consistency model can be assumed, some programs escape from the SC constraints for performance reasons. In this document we propose an approach for verifying programs under the TSO memory consistency model.

1 Introduction

Memory Consistency. In the past, most verification approaches for concurrent programs have assumed that memory behaves as if all threads' memory accesses are executed in an interleaved fashion directly on a single global memory, such that there is a total order on all memory accesses and each read operation on a memory location yields the value of the most recent preceding write operation to that memory location in that total order. This memory consistency model is known as *sequential consistency* (SC).

However, real programming platforms (hardware architectures and compiled or interpreted programming languages) do not offer simple sequential consistency: these platforms' memory access primitives have weaker memory consistency semantics, in order to allow for important performance optimizations at the level of the memory hierarchy, the processor, and any compilation steps, such as caching, pipelining, and common subexpression elimination.

Still, most programming languages guarantee that programs for which all sequentially consistent executions are *data-race-free*, in some sense, have only sequentially consistent executions. Data-race-freedom usually means that all conflicting memory accesses are ordered by a *happens-before* relation induced by program order and by synchronization constructs such as mutual exclusion locks. As a result, most programmers can indeed safely assume sequential consistency and use the program verification techniques which assume sequential consistency and which verify data-race-freedom.

Nonetheless, in some cases it is necessary to consider non-data-race-free code, in order to avoid the performance overhead of synchronization. For such programs, reasoning must occur directly in terms of the weaker memory consistency semantics offered by the platform.

One of the most important programming platforms today is the x86/x64 family of processor architectures. The memory consistency model offered by these architectures is the x86-TSO (Total Store Order) memory model. In the TSO memory model, each thread has a *write buffer*. Write operations by a thread are enqueued at the end of its write buffer; the memory subsystem decides arbitrarily when to dequeue a write operation from the front of a thread's write buffer and apply it to main memory. Main memory itself is sequentially consistent. Read operations by a thread are satisfied from the thread's write buffer, if possible, or else from main memory.

The Approach. In this document, we propose a verification approach for programs that use memory accesses with TSO semantics. To motivate and illustrate the approach, we will use the running example of a Java virtual machine implementation for a TSO platform. Such an implementation must implement Java field accesses efficiently. Assume it implements them as plain TSO read and write operations. A challenge is the issue of object initialization: threads may allocate Java objects concurrently, and leak references to those objects into shared fields without any synchronization. A thread that accesses an object expects the object to contain a valid pointer to a virtual method dispatch table (*vtable*), even if the object was obtained through a race. On a TSO platform, this is easy to achieve by making sure that an object's vtable pointer is initialized before references to the object are stored into fields of existing objects in the heap. The FIFO nature of the write buffers then ensures that if a thread sees an object reference, it also sees the properly initialized vtable pointer.

Notice that this program is not data-race-free, and therefore program verification approaches that verify data-race-freedom are not applicable. Instead, we propose an extension of separation logic with *TSO spaces*. TSO spaces are similar to the shared resources of Concurrent Separation Logic (CSL), except that memory owned by a TSO space may be accessed through TSO operations (memory accesses with TSO semantics) rather than classical critical sections. Also, where CSL associates a resource invariant with each resource, we associate an *abstract state space* with each TSO space, as well as an *abstraction predicate* that associates each abstract state with a corresponding separation logic assertion. The abstract state space is equipped with an *abstract reachability* pre-order $\preceq$.

Knowledge about the state of a TSO space in the proof of a thread is represented as an abstract state, representing the thread's view of the state of the TSO space. A thread's view is a lower bound (under the abstract reachability pre-order) on the actual state of the TSO space.

Each TSO write operation is associated with an *abstract state transition function*, mapping an abstract pre-state to an abstract post-state. These abstract state transition functions must respect the abstract reachability relation and properly abstract the concrete behavior of the TSO operation. Also, crucially, to account for TSO's relaxed behavior, the abstract state transition functions must be *monotonic*: they must respect reachability and be sound with respect to concrete behavior not only in the current abstract state, but also in all reachable future abstract states. In a thread's proof, when the thread performs a TSO write operation, its local view of the TSO space is updated per the abstract state transition function.

Each TSO read operation is associated with a function f mapping result values to new abstract states. This function must satisfy the property that for any result value v, and for any future abstract state α', if the target location may have value v in this abstract state, then $f(v) \preceq \alpha'$. In a thread's proof, when the thread performs a TSO read operation, its local view of the TSO space is replaced with the (hopefully more precise) lower bound given by function f.

Verifying the Example. For example, to verify the Java virtual machine implementation using the proposed logic, we put the heap in a TSO space. As the abstract state space, we adopt the powerset of addresses in this heap, each abstract state representing the set of the addresses of the currently allocated and initialized objects. The subset relation serves as the reachability order. The abstraction predicate states 1) that allocated objects occupy disjoint heap space, 2) that they properly point to an existing virtual method dispatch table, and 3) that their fields point to allocated objects. Each thread is aware of the addresses of objects it allocated itself, as well as addresses read from fields of known objects. That is, reading a field updates the thread's view by inserting the newly discovered object address into the abstract state. After a thread allocates an object and initializes its run-time type information, it performs a no-op operation on the TSO space to update its local view of the set of allocated objects, inserting the newly allocated object into the abstract state. Writing the value of a local variable to a field corresponds to the identity function at the abstract level, since all object references a thread holds in local variables are already in the thread's local view.

The remainder of this document is structured as follows. In Section 2, we present the basic idea of the approach in the context of a simplified programming language without pointers. In Sec 3, we extend the programming language with locked instructions. In Sections 5 and 6 we integrate our approach into separation logic. In Section 7 we report on a preliminary encoding of the approach into the logic of the VeriFast program verifier. We offer a conclusion in Section 8.

2 The Basic Idea

2.1 Program Syntax

We consider a simple programming language with threads and shared global variables $g \in G$. There are no local variables.

$$\begin{array}{rcl} \mathit{Heaps} & = & G \to \mathbb{Z} \\ \delta \in \Delta & = & \mathit{Heaps} \to \mathit{Heaps} \\ c \in \mathit{Cmds} & ::= & \delta; c \mid c(g) \mid \textbf{done} \mid \textbf{fail} \\ \mathit{prog} & ::= & c \mid\mid \cdots \mid\mid c \end{array}$$

The heaps $h \in \mathit{Heaps}$ are the maps from variable names to values. A *command* is either an *update* δ (a function from heaps to heaps) followed by another command; or a *read operation* $c(g)$ consisting of the variable g to be read, and a function $c(\cdot)$ from values to commands; or the operation **done** indicating the end of the thread; or the operation **fail** indicating a failure. A program is a parallel composition of commands.

We consider generic updates instead of writes of individual variables, to allow for the instrumentation of writes with *ghost updates*.

2.2 Small-Step Semantics

The semantics of the programming language is defined as a small-step relation $\leadsto$ between machine configurations $\gamma \in \mathsf{Configs}$. A machine configuration consists of a heap and a multiset of thread configurations $\theta \in \mathit{ThreadConfigs}$. (A multiset over elements of a set X can be modeled as a function that maps each $x \in X$ to the number of times it occurs in the multiset.) A thread configuration consists of a write buffer (a sequence of updates) and a command.

$$\begin{array}{rcl} \mathit{Buffers} & = & \Delta^* \\ \theta \in \mathit{ThreadConfigs} & = & \mathit{Buffers} \times \mathit{Cmds} \\ \gamma \in \mathit{Configs} & = & \mathit{Heaps} \times (\mathit{ThreadConfigs} \rightarrow_{\mathrm{fin}} \mathbb{N}) \\ \leadsto & \subseteq & \mathit{Configs} \times \mathit{Configs} \end{array}$$

There are only three kinds of steps: an ENQUEUE step enqueues an update; a READ step reduces a read operation; and a DEQUEUE step applies an update to the heap.

$$\textsc{Enqueue}\ \frac{}{(h, \{\!\!\{(\overline{\delta}, \delta; c)\}\!\!\} \uplus \Theta) \leadsto (h, \{\!\!\{(\overline{\delta} \cdot \delta, c)\}\!\!\} \uplus \Theta)}$$

$$\textsc{Read}\ \frac{}{(h, \{\!\!\{(\overline{\delta}, c(g))\}\!\!\} \uplus \Theta) \leadsto (h, \{\!\!\{(\overline{\delta}, c(\overline{\delta}(h)(g)))\}\!\!\} \uplus \Theta)}$$

$$\textsc{Dequeue}\ \frac{}{(h, \{\!\!\{(\delta \cdot \overline{\delta}, c)\}\!\!\} \uplus \Theta) \leadsto (\delta(h), \{\!\!\{(\overline{\delta}, c)\}\!\!\} \uplus \Theta)}$$

Definition 1 (Failing Configuration)**.** *We say a machine configuration γ is* failing, *denoted $\gamma \in \mathit{Fail}$, if there exists $h, \overline{\delta}, \Theta$ such that $\gamma = (h, \{\!\!\{(\overline{\delta}, \mathbf{fail})\}\!\!\} \uplus \Theta)$.*

Definition 2 (Program Safety)**.** *A program $c_1 \mid\mid \cdots \mid\mid c_n$ is* safe *when started from an initial heap h_0 if $\forall \gamma'.\ (h, \{\!\!\{(\epsilon, c_1)\}\!\!\} \uplus \cdots \uplus \{\!\!\{(\epsilon, c_n)\}\!\!\}) \leadsto^* \gamma' \Rightarrow \gamma' \notin \mathit{Fail}$.*

2.3 Proof System

We assume a set $A \subseteq 2^{\mathit{Heap}}$ of heap predicates, *the abstract state space*, to be chosen by the proof developer. We use α to range over A. We assume an *abstract reachability pre-order* $\preceq$ on A.

We assume a partitioning of the variables $g \in G$ into the shared variables G_{s} and the *owned variables* of thread i G_i, for each i. We assume that only thread i updates the variables owned by thread i; however, reading is not restricted. A heap predicate $L \subseteq \mathit{Heaps}$ is *local to thread i* if it can be invalidated only by updates to variables owned by thread i: $\mathit{Local}_i = \{L \mid \forall h, h'.\ h \in L \wedge h|_{G_i} = h'|_{G_i} \Rightarrow h' \in L\}$.

We define a validity condition on commands, $\mathsf{valid}_i(\alpha, L, c)$, where $\alpha \in A$ is thread i's local view of the heap (including the shared variables), $L \in \mathit{Local}_i$ is information about thread i's owned variables, and $c \in \mathit{Cmds}$ is the command being executed by the thread.

$$
\begin{array}{l}
\mathsf{valid}_i(\alpha, L, \delta; c) = \\
\quad \exists f \in A \rightarrow A, L' \in \mathit{Local}_i. \\
\qquad (\forall \alpha' \succeq \alpha.\ f(\alpha') \succeq \alpha') \wedge \\
\qquad (\forall \alpha' \succeq \alpha, \alpha'' \succeq \alpha'.\ f(\alpha'') \succeq f(\alpha')) \wedge \\
\qquad (\forall \alpha' \succeq \alpha, h \in \alpha' \cap L.\ \delta(h) \in f(\alpha') \cap L') \wedge \\
\qquad \mathsf{valid}_i(f(\alpha), L', c) \\
\mathsf{valid}_i(\alpha, L, c(g)) = \\
\quad \forall v.\ \exists \alpha'. \\
\qquad (\forall \alpha'' \succeq \alpha, h \in \alpha'' \cap L.\ h(g) = v \Rightarrow \alpha'' \succeq \alpha') \wedge \\
\qquad \mathsf{valid}_i(\alpha', L, c(v)) \\
\mathsf{valid}_i(\alpha, L, \mathbf{done}) = \mathsf{True} \\
\mathsf{valid}_i(\alpha, L, \mathbf{fail}) = \forall \alpha' \succeq \alpha, h \in \alpha' \cap L.\ \mathsf{False}
\end{array}
$$

An update is valid if there exists an abstract version f of the update and a local postcondition L' such that in all abstract states α' reachable from the thread's view α, f respects abstract reachability (first conjunct), f is monotonic (second conjunct), and f soundly abstracts the update (third conjunct) given the local precondition L and the local postcondition L'. Furthermore, the continuation must be valid under the post-abstract state $f(\alpha)$ and the local postcondition L'.

As will become clear from the soundness proof, the monotonicity requirement ensures that the current thread's proof outline remains valid when updates of other threads are dequeued while the current thread's write buffer is nonempty.

A read operation is valid if for each possible result of the read operation, there exists a new view α' such that all abstract states α'' reachable from the current view where the result is possible (given local information L) are reachable from α', and furthermore the continuation is valid under this updated view α'.

A **done** operation is always valid. A **fail** operation is valid provided no abstract state reachable from the current view is feasible.

Definition 3 (Program Validity). *We say a program $c_1 \mid\mid \cdots \mid\mid c_n$, started from initial heap h_0, is* valid *if there exists an initial abstract state $\alpha \in A$ and local preconditions $L_i \in \mathit{Local}_i$ such that $h \in \alpha \cap L_1 \cap \cdots \cap L_n$ and $\mathsf{valid}_i(\alpha, L_i, c_i)$, for all i.*

2.4 Soundness

The target soundness property is that valid programs are safe. In the remainder of this subsection, we sketch a proof of this property.

We define validity of a thread configuration $\mathsf{valid\_tcfg}_i(\alpha, L, \theta)$ as follows:

$$\mathsf{valid\_tcfg}_i(\alpha, L, (\delta_1 \cdots \delta_n, c)) = \mathsf{valid}_i(\alpha, L, \delta_1; \cdots; \delta_n; c)$$

We define validity of a machine configuration $\mathsf{valid\_cfg}(\gamma)$ as follows:

$$
\begin{array}{l}
\mathsf{valid\_cfg}((h, \{\!\!\{\theta_1, \ldots, \theta_n\}\!\!\})) = \\
\quad \exists \alpha, L_1, \ldots, L_n.\ h \in \alpha \cap L_1 \cap \cdots \cap L_n \wedge \forall i.\ \mathsf{valid\_tcfg}_i(\alpha, L_i, \theta_i)
\end{array}
$$

where $L_i \in \mathit{Local}_i$.

Lemma 1. *If $\mathsf{valid}_i(\alpha, L, c)$ and $\alpha' \succeq \alpha$ then $\mathsf{valid}_i(\alpha', L, c)$.*

Proof. By induction on c. □

Lemma 2. *If* $\mathsf{valid\_cfg}(\gamma)$ *and* $\gamma \rightsquigarrow \gamma'$*, then* $\mathsf{valid\_cfg}(\gamma')$.

Proof. By case analysis on the step rule.

- **Case** Enqueue. Trivial.
- **Case** Read. Assume $\gamma = (h, \{\!\!\{(\delta_1 \cdot \cdots \cdot \delta_m, c(g))\}\!\!\} \uplus \Theta)$ and $h'(g) = v$ with $h' = (\delta_m \circ \cdots \circ \delta_1)(h)(g) = v$ and $\gamma' = (h, \{\!\!\{(\delta_1 \cdot \cdots \cdot \delta_m, c(v))\}\!\!\} \uplus \Theta)$. By the first premise, there exists an α and an L_i^0 such that $h \in \alpha \cap L_i^0$ and $\mathsf{valid}_i(\alpha, L_i^0, \delta_1; \ldots; \delta_m; c(g))$. By definition of valid_i, there exist f_j and L_i^j, for $j \in \{1, \ldots, m\}$, such that $\mathsf{valid}_i(\alpha'', L_i^j, c(g))$ with $\alpha'' = (f_m \circ \cdots \circ f_1)(\alpha)$. Note that $\alpha'' \succeq \alpha$ (by the first conjunct of validity of updates) and $h' \in \alpha''$ (by the third conjunct of validity of updates) and $h''(g) = v$. Therefore, by validity of reads, there exists an $\alpha' \preceq \alpha''$ such that $\mathsf{valid}_i(\alpha', L_i^m, c(v))$, and thus, by Lemma 1, $\mathsf{valid}_i(\alpha'', L_i^m, c(v))$.
- **Case** Dequeue. We need to prove three things: that the new heap satisfies the new abstract state and all local preconditions; that the new configuration of the thread whose write was dequeued is valid; and that all other threads' configurations remain valid. First of all: we know $h \in \alpha \cap L_i$ and therefore $\delta(h) \in f(\alpha) \cap L_i'$. It follows directly that the new configuration of the active thread is valid. The other threads' local preconditions are preserved, by their locality. The other threads' configurations remain valid by Lemma 1.

□

Lemma 3. *If* $\mathsf{valid\_cfg}(\gamma)$ *then* $\gamma \notin \mathit{Fail}$.

Proof. Assume $h \in \alpha \cap L_1 \cap \cdots \cap L_n \wedge \forall i.\ \mathsf{valid\_tcfg}_i(\alpha, L_i, \theta_i)$. Assume $\theta_i = (\overline{\delta}, \mathbf{fail})$ for some i. Then by $\mathsf{valid\_tcfg}_i(\alpha, L_i, \theta_i)$ we have $\forall \alpha' \succeq \alpha, h \in \alpha' \cap L_i$. False. By taking $\alpha' = \alpha$ we obtain a contradiction. □

2.5 Examples

Virtual Machine. We encode a simplified version of the virtual machine example into our formal programming language.

Firstly, we encode addressable memory into the language by choosing an indexed set of variables: let $G = \{\mathsf{m}_0, \ldots, \mathsf{m}_{9999}\}$.

We consider "objects" consisting just of a single field; we do not consider vtable pointers in this example. Each field of each allocated object must point to an allocated object. In the initial heap, there is a single object at address 0, and its field points to itself. All other memory locations hold the value -1, which is not an address: $h_0 = \{\mathsf{m}_0 \mapsto 0, \mathsf{m}_{1\ldots9999} \mapsto -1\}$.

The program consists of two threads. The first thread initializes an object at location 1, by storing a reference to itself into its field, and then publishes this object by storing a reference to the object into the field of the initially allocated object at location 0. The second thread reads the field of the initial object and asserts that the value is an address. It then reads the field of the object at this address and asserts that the resulting value is again an address.

$c_1 = \langle \mathsf{m}_1 := 1 \rangle; \langle \mathsf{m}_0 := 1 \rangle; \mathbf{done}$
$c_2 = \ell \leftarrow \mathsf{m}_0; \mathsf{assert}(0 \leq \ell \leq 9999); \ell' \leftarrow \mathsf{m}_\ell; \mathsf{assert}(0 \leq \ell' \leq 9999); \mathbf{done}$

Notice that this program relies on the FIFO nature of the write buffers in the TSO memory model.

We can verify this program in the approach of this section as follows. As the abstract state space, we take the predicates α_Λ where Λ is a set of integers, and α_Λ is satisfied by a heap h if each value in Λ is an address that is mapped by h to a value in Λ:

$$A = \{\alpha_\Lambda \mid \alpha_\Lambda = \alpha_\Lambda = \{h \mid \forall \ell \in \Lambda.\ \ell \in \mathit{Addresses} \land h(\mathsf{m}_\ell) \in \Lambda\}\}$$

where $\mathit{Addresses} = \{0, \ldots, 9999\}$.

Abstract reachability corresponds to the subset relation on the indices: $\alpha_\Lambda \preceq \alpha_{\Lambda'} \Leftrightarrow \Lambda \subseteq \Lambda'$.

In this proof, we do not use the local predicates. That is, we take $L = \mathit{Heaps}$ for all local preconditions and postconditions L.

Our initial abstract state is $\alpha_{\{0\}}$.

The proof outline for c_1 is as follows:[1]

$$\begin{array}{ll}
\alpha_{\{0\}} & \\
\langle \mathsf{m}_1 := 1 \rangle; & f(\alpha_\Lambda) = \alpha_{\Lambda \cup \{1\}} \\
\alpha_{\{0,1\}} & \\
\langle \mathsf{m}_0 := 1 \rangle; & f(\alpha_\Lambda) = \alpha_\Lambda \\
\alpha_{\{0,1\}} & \\
\mathbf{done} &
\end{array}$$

That is, the first update is associated with a function that adds the value 1 to Λ. The second update is associated with the identity function. It's easy to see that this proof outline is valid.

The proof outline for c_2 is as follows. (Notice that $\Lambda \not\subseteq \mathit{Addresses} \Rightarrow \alpha_\Lambda = \emptyset$; i.e., if an abstract state's index contains values that are not addresses, the abstract state is infeasible.)

$$\begin{array}{ll}
\alpha_{\{0\}} & \\
\ell \leftarrow \mathsf{m}_0; & \alpha' = \alpha_{\{0,\ell\}} \\
\alpha_{\{0,\ell\}} & \\
\mathsf{assert}(0 \leq \ell \leq 9999); & \\
\alpha_{\{0,\ell\}} & \\
\ell' \leftarrow \mathsf{m}_\ell; & \alpha' = \alpha_{\{0,\ell,\ell'\}} \\
\alpha_{\{0,\ell,\ell'\}} & \\
\mathsf{assert}(0 \leq \ell' \leq 9999); & \\
\alpha_{\{0,\ell,\ell'\}} & \\
\mathbf{done} &
\end{array}$$

[1] In these proof outlines, we specify the abstract state before and after each program command; furthermore, we annotate each update with abstract version (the f function), and each read operation with its new abstract state α' (which may depend on the result of the read operation).

Producer-Consumer. For this example, consider two threads that communicate via a single shared variable b. This variable (the *buffer*) is either *empty*, if its value is 0, or *full* otherwise. The producer thread puts the integers from k down to 1 into the buffer. Putting a value into the buffer means writing it into the buffer and then waiting for the buffer to become empty. The consumer thread repeatedly takes a value from the buffer and asserts that it is the next lower integer. Taking a value from the buffer means reading the buffer until a nonzero value is read, and then writing zero.

$\mathsf{prod}_0 = \textbf{done}$
$\mathsf{prod}_{k+1} = \langle \mathsf{b} := k+1 \rangle; v \leftarrow \mathsf{b}; \textbf{if } v = 0 \textbf{ then } \mathsf{prod}_k \textbf{ else done}$
$\mathsf{cons}_0 = \textbf{done}$
$\mathsf{cons}_{k+1} = v \leftarrow \mathsf{b}; \textbf{if } v = 0 \textbf{ then done else } (\textbf{assert } v = k+1; \langle \mathsf{b} := 0 \rangle; \mathsf{cons}_k)$

Our verification goal is to prove that $\forall k.\ \mathsf{prod}_k \mid\mid \mathsf{cons}_k$ is safe when started in heap $\{\mathsf{b} \mapsto 0\}$.

We introduce two ghost variables: p and c, both initially equal to $k+1$. p is owned by the producer, and c is owned by the consumer.

We instrument the program with ghost updates:

$\mathsf{prod'}_0 = \textbf{done}$
$\mathsf{prod'}_{k+1} = \langle \mathsf{b} := k+1; \mathsf{p} := k+1 \rangle; v \leftarrow \mathsf{b}; \textbf{if } v = 0 \textbf{ then } \mathsf{prod'}_k \textbf{ else done}$
$\mathsf{cons'}_0 = \textbf{done}$
$\mathsf{cons'}_{k+1} =$
  $v \leftarrow \mathsf{b};$
  $\textbf{if } v = 0 \textbf{ then done else } (\textbf{assert } v = k+1; \langle \mathsf{b} := 0; \mathsf{c} := k+1 \rangle; \mathsf{cons'}_k)$

For the proof, we define the abstract state space as $A = \{\alpha_{p,c} \mid \alpha_{p,c} = \{\{\mathsf{p} \mapsto p, \mathsf{c} \mapsto c, \mathsf{b} \mapsto b \mid p = c \land b = 0 \lor p = c-1 \land b = p\}\}\}$. That is, each abstract state $\alpha_{p,c}$ is either a singleton (if $p \le c \le p+1$) or the empty set (otherwise).

We define the abstract reachability relation $\alpha_{p,c} \preceq \alpha_{p',c'}$ as $p > p' \lor p = p' \land c \ge c'$.

The initial abstract state is $\alpha_{k+1,k+1}$.

We use the local predicates $P_k = \{h \mid h(\mathsf{p}) = k\}$ and $C_k = \{h \mid h(\mathsf{c}) = h\}$.

We prove by induction on k that $\mathsf{valid}_1(\alpha_{k+1,k+1}, P_{k+1}, \mathsf{prod}_k)$. Case $k = 0$ is trivial. Assume $k > 0$.

$\alpha_{k+1,k+1}, P_{k+1}$
$\langle \mathsf{b} := k; \mathsf{p} := k \rangle;\qquad f(\alpha_{p,c}) = \alpha_{p-1,c}$
$\alpha_{k,k+1}, P_k$
$v \leftarrow \mathsf{b};\qquad \alpha' = (v = 0\ ?\ \alpha_{k,k}\ :\ \alpha_{k,k+1})$
$v = 0\ ?\ \alpha_{k,k}, P_k\ :\ \alpha_{k,k+1}, P_k$
$\textbf{if } v = 0 \textbf{ then } \mathsf{prod'}_{k-1} \textbf{ else done}$

We prove by induction on k that $\mathsf{valid}_2(\alpha_{k+1,k+1}, C_{k+1}, \mathsf{cons}_k)$. Case $k = 0$

is trivial. Assume $k > 0$.

```
$\alpha_{k+1,k+1}, C_{k+1}$
$v \leftarrow \mathsf{b};$        $\alpha' = (v = 0\ ?\ \alpha_{k+1,k+1}\ :\ \alpha_{k,k+1})$
$v = 0\ ?\ \alpha_{k+1,k+1}, C_{k+1}\ :\ \alpha_{v,v+1}, C_{k+1}$
if $v = 0$ then done else (
  $\alpha_{v,v+1}, C_{k+1}$
  assert $v = k$;
  $\alpha_{k,k+1}, C_{k+1}$
  $\langle \mathsf{b} := 0; \mathsf{c} := k \rangle;$        $f(\alpha_{p,c}) = \alpha_{p,c-1}$
  $\alpha_{k,k}, C_k$
  $\mathsf{cons'}_{k-1}$
)
```

3 Locked Instructions

The x86 architecture supports *locked instructions*, i.e. machine instructions prefixed by the `LOCK` prefix. The semantics of the `LOCK` prefix is that during execution of the prefixed instruction, main memory is locked, so no other threads' write operations are dequeued from their write buffers; furthermore, after the instruction is finished but before the lock is released, the current thread's write buffer is flushed.

We extend our programming language with locked operations:

$$c ::= \cdots \mid \mathbf{locked}\ \delta; c(\mathsf{heap})$$

A locked operation consists of an update δ and a continuation c parameterized by a heap (i.e. a function from heaps to commands).

We extend the program semantics with a single step rule:

$$\textsc{Locked}\quad \frac{}{(h, \{\!\{(\epsilon, \mathbf{locked}\ \delta; c(\mathsf{heap}))\}\!\} \uplus \Theta) \rightsquigarrow (\delta(h), \{\!\{(\epsilon, c(h))\}\!\} \uplus \Theta)}$$

We define validity of a locked operation:

$$\begin{array}{l} \mathsf{valid}_i(\alpha, L, \mathbf{locked}\ \delta; c(\mathsf{heap})) = \\ \quad \forall \alpha' \succeq \alpha, h \in \alpha' \cap L.\ \exists \alpha'' \succeq \alpha', L' \in \mathit{Local}_i. \\ \quad\quad \delta(h) \in \alpha'' \cap L' \wedge \mathsf{valid}_i(\alpha'', L', c(h)) \end{array}$$

We extend the soundness proof.

Theorem 1. *The extended system is sound.*

Proof. Lemma 1 is preserved trivially. For Lemma 2, the proof is analogous to case Dequeue. □

Example: Mutex. Consider a program that protects a shared resource using a mutual exclusion lock implemented using a Compare-And-Set (CAS) instruction. The mutex is implemented as a shared variable m whose value is zero when the mutex is not held, and one when the mutex is held. The shared resource is a shared variable r whose value should always be 0 except during an operation on the resource.

Each thread attempts to acquire the resource by performing a CAS instruction on m, attempting to set it from 0 to 1. If successful, it asserts that r equals 0, then sets it to 1, then resets it to 0, and finally releases the mutex by setting m to 0.

The command executed by each thread is as follows:

```
$\mathsf{thread}_i =$
  $h \leftarrow$ **locked** $\langle$**if** $\mathsf{m} = 0$ **then** $\mathsf{m} := 1\rangle$;
  **assume** $h(\mathsf{m}) = 0$;
  $r \leftarrow \mathsf{r}$; **assert** $r = 0$;
  $\langle \mathsf{r} := 1\rangle; \langle \mathsf{r} := 0\rangle$;
  $\langle \mathsf{m} := 0\rangle$
```

The initial heap h_0 is $\{\mathsf{m} \mapsto 0, \mathsf{r} \mapsto 0\}$.

We introduce one shared ghost variable u_0, and one ghost variable u_i for each thread i, owned by thread i, with $i > 0$. Initially u_0 has value 1 and all other u_i have value 0.

We instrument the program with ghost updates as follows:

```
$\mathsf{thread}'_i =$
  $h \leftarrow$ **locked** $\langle$**if** $\mathsf{m} = 0$ **then** $(\mathsf{m} := 1; \mathsf{u}_i, \mathsf{u}_0 := \mathsf{u}_0, 0)\rangle$;
  **assume** $h(\mathsf{m}) = 0$;
  $r \leftarrow \mathsf{r}$; **assert** $r = 0$;
  $\langle \mathsf{r} := 1; \mathsf{u}_i{+}{+}\rangle; \langle \mathsf{r} := 0; \mathsf{u}_i{+}{+}\rangle$;
  $\langle \mathsf{m} := 0; \mathsf{u}_0, \mathsf{u}_i := \mathsf{u}_i, 0\rangle$
```

We take as the abstract state space the set $A = \{\alpha_{u,r} \mid \alpha_{u,r} = \{h \mid h(\mathsf{r}) = r \wedge \exists i.\ (i \neq 0 \wedge h(\mathsf{m}) = 1 \vee i = 0 \wedge r = 0 \wedge h(\mathsf{m}) = 0) \wedge \forall j.\ j = i \wedge h(\mathsf{u}_j) = u \vee j \neq i \wedge h(\mathsf{u}_j) = 0\}\}$. In each abstract state, the current update count u is stored either in u_0, indicating that the mutex is not held by any thread, or in u_i with $i > 0$, indicating that the mutex is held by thread i.

We define the abstract reachability relation $\alpha_{u,r} \preceq \alpha_{u',r'}$ as $u < u' \vee (u, r) = (u', r')$.

We define local predicates $L_{i,u} = \{h \mid h(\mathsf{u}_i) = u\}$ and $L_\top = \mathit{Heaps}$.

The initial abstract state is $\alpha_{1,0}$.

Proof outline:

```
$\alpha_{1,0}, L_\top$
$h \leftarrow$ **locked** $\langle$**if** $\mathsf{m} = 0$ **then** $(\mathsf{m} := 1; \mathsf{u}_i, \mathsf{u}_0 := \mathsf{u}_0, 0)\rangle$;
$h(\mathsf{m}) = 0\ ?\ \alpha_{h(\mathsf{u}_0),0}, L_{i,h(\mathsf{u}_0)}\ :\ \alpha_{1,0}$
**assume** $h(\mathsf{m}) = 0$;
$\alpha_{h(\mathsf{u}_0),0}, L_{i,h(\mathsf{u}_0)}$
$r \leftarrow \mathsf{r}$;        $\alpha' = (r = 0\ ?\ \alpha_{h(\mathsf{u}_0),0}\ :\ \alpha_{h(\mathsf{u}_0)+1,r})$
**assert** $r = 0$;
$\alpha_{h(\mathsf{u}_0),0}, L_{i,h(\mathsf{u}_0)}$
$\langle \mathsf{r} := 1; \mathsf{u}_i{+}{+}\rangle$;        $f(\alpha_{u,r}) = \alpha_{u+1,1}$
$\alpha_{h(\mathsf{u}_0)+1,1}, L_{i,h(\mathsf{u}_0)+1}$
$\langle \mathsf{r} := 0; \mathsf{u}_i{+}{+}\rangle$;        $f(\alpha_{u,r}) = \alpha_{u+1,0}$
$\alpha_{h(\mathsf{u}_0)+2,0}, L_{i,h(\mathsf{u}_0)+2}$
$\langle \mathsf{m} := 0; \mathsf{u}_0, \mathsf{u}_i := \mathsf{u}_i, 0\rangle$        $f(\alpha_{u,r}) = \alpha_{u,r}$
$\alpha_{h(\mathsf{u}_0)+2,0}, L_\top$
```

4 Marriage of TSO and Separation Logic

In the next two sections, we will marry our approach with ownership and the assertion logic of separation logic on the one hand, and with the Hoare logic-style proofs approach on the other hand.

5 Marriage of TSO, Ownership, and Separation Assertions

In the preceding sections, we assumed a static partitioning of the set of variables into shared variables and variables owned by particular threads. In this section, we adopt a more dynamic approach by considering *fractional heaps*. The fractional heaps $H \in \mathit{FracHeaps}$ are defined as follows:

$$\mathit{FracHeaps} = G \rightharpoonup \mathbb{Z} \times (0, 1]$$

For $H, H_1, H_2 \in \mathit{FracHeaps}$, we say $H = H_1 \bullet H_2$ iff

$$\forall g, v.\ H(g, v) = H_1(g, v) + H_2(g, v)$$

where

$$H(g, v) = \begin{cases} \pi & \text{if } H(g) = (v, \pi) \\ 0 & \text{if } \nexists\pi.\ H(g) = (v, \pi) \end{cases}$$

We reinterpret updates of the programming language as follows:

$$\delta(H) = H' \Leftrightarrow \forall H_0, h.\ H \bullet H_0 = h \Rightarrow \delta(h) = H' \bullet H_0$$

We now pick both the abstract states and the local predicates from the powerset of fractional heaps: $A \subseteq 2^{\mathit{FracHeaps}}$, $L_i \subseteq \mathit{FracHeaps}$.

Validity of commands is updated as follows:

```
$\mathsf{valid}(\alpha, L, \delta; c) =$
  $\exists f \in A \to A, L' \subseteq \mathit{FracHeaps}.$
    $(\forall \alpha' \succeq \alpha.\ f(\alpha') \succeq \alpha') \wedge$
    $(\forall \alpha' \succeq \alpha, \alpha'' \succeq \alpha'.\ f(\alpha'') \succeq f(\alpha')) \wedge$
    $(\forall \alpha' \succeq \alpha, H \in \alpha' \bullet L.\ \delta(H) \in f(\alpha') \bullet L') \wedge$
    $\mathsf{valid}(f(\alpha), L', c)$
$\mathsf{valid}(\alpha, L, c(g)) =$
  $(\forall v, H \in L.\ g \in \mathrm{dom}(H) \wedge (H(g) = (v, \_) \Rightarrow \mathsf{valid}(\alpha, L, c(v)))) \vee$
  $\forall v.\ \exists \alpha'.$
    $\forall \alpha'' \succeq \alpha, H \in \alpha'' \bullet L.\ g \in \mathrm{dom}(H) \wedge (H(g) = (v, \_) \Rightarrow \alpha'' \succeq \alpha') \wedge$
    $\mathsf{valid}(\alpha', L, c(v))$
$\mathsf{valid}(\alpha, L, \mathbf{done}) = \mathsf{True}$
$\mathsf{valid}(\alpha, L, \mathbf{fail}) = \forall \alpha' \succeq \alpha, h \in \alpha' \bullet L.\ \mathsf{False}$
$\mathsf{valid}(\alpha, L, \mathbf{locked}\ \delta; c(\mathsf{heap})) =$
  $\forall \alpha' \succeq \alpha, H \in \alpha' \bullet L.\ \exists \alpha'' \succeq \alpha', L'.$
    $\delta(H) \in \alpha'' \bullet L' \wedge \forall H_0, h.\ h = H \bullet H_0 \Rightarrow \mathsf{valid}(\alpha'', L', c(h))$
```

Notice that mostly we simply replaced occurrences of $\alpha \cap L$ by $\alpha \bullet L$ to reflect the fact that α and L are now predicates over fractional heaps. We also added an alternative clause for validity of read operations, allowing for slightly simpler proofs in the case where certain result values can be excluded based on local knowledge and no refinement of the abstract state is required.

Validity of a configuration is now defined as:

$$\begin{array}{l}\mathsf{valid\_cfg}((h, \{\!\!\{\theta_1, \ldots, \theta_n\}\!\!\})) = \\ \quad \exists \alpha, L_1, \ldots, L_n, h \in \alpha \bullet L_1 \bullet \cdots \bullet L_n.\ \forall i.\ \mathsf{valid}(\alpha, L_i, \theta_i)\end{array}$$

where $P \bullet Q = \{H \mid \exists H_1 \in P, H_2 \in Q.\ H = H_1 \bullet H_2\}$.

Theorem 2. *This approach is sound.*

Proof. Monotonicity is preserved. Preservation of configuration validity under the step relation is entirely analogous; the only difference now is that, when an update of thread i is dequeued, preservation of L_j for $j \neq i$ is argued based on the fact that $\delta(H)$ is well-defined and therefore $\delta(H \bullet H_{\mathrm{L}}) = \delta(H) \bullet H_{\mathrm{L}}$. The case where the step is a read operation and the new clause for validity of read operations is used, is discharged easily. ☐

Example. Exploiting separation logic, we can simplify the proof of the mutex example of the preceding section.

We recall that the command executed by each thread is as follows:

```
$\mathsf{thread}_i =$
  $h \leftarrow$ **locked** $\langle$**if** $\mathsf{m} = 0$ **then** $\mathsf{m} := 1\rangle$;
  **assume** $h(\mathsf{m}) = 0$;
  $r \leftarrow \mathsf{r}$; **assert** $r = 0$;
  $\langle \mathsf{r} := 1\rangle; \langle \mathsf{r} := 0\rangle$;
  $\langle \mathsf{m} := 0\rangle$
```

For this version of the proof, we do not need to introduce any ghost variables. Indeed, in this version the *permissions* play the corresponding role.

The abstract state space is a singleton: $A = \{\alpha\}$; $\alpha = \{H \mid H \vDash \mathsf{m} \mapsto 0 * \mathsf{r} \mapsto 0 \vee \mathsf{m} \mapsto 1\}$.

For the local predicates, we use separation logic assertions.

Proof outline:

```
$\alpha$, **emp**
$h \leftarrow$ **locked** $\langle$**if** $\mathsf{m} = 0$ **then** $\mathsf{m} := 1\rangle$;
$h(\mathsf{m}) = 0\ ?\ \alpha, \mathsf{r} \mapsto 0\ :\ \alpha,$ **emp**
**assume** $h(\mathsf{m}) = 0$;
$\alpha, \mathsf{r} \mapsto 0$
$r \leftarrow \mathsf{r}$;
$\alpha, \mathsf{r} \mapsto 0 \wedge r = 0$
**assert** $r = 0$;
$\alpha, \mathsf{r} \mapsto 0$
$\langle \mathsf{r} := 1\rangle$;        $f(\alpha) = \alpha$
$\alpha, \mathsf{r} \mapsto 1$
$\langle \mathsf{r} := 0\rangle$;        $f(\alpha) = \alpha$
$\alpha, \mathsf{r} \mapsto 0$
$\langle \mathsf{m} := 0\rangle$         $f(\alpha) = \alpha$
$\alpha$, **emp**
```

6 A Hoare Logic for TSO

In this section we adopt a more realistic programming language, and we define a Hoare logic for it, based on separation logic.

6.1 Syntax and Semantics of Programs

The syntax of the programming language is as follows:

$$\begin{array}{rl} e ::= & x \mid z \mid e + e \mid e - e \\ b ::= & e = e \mid e < e \mid \neg b \\ u ::= & x := [e] \mid [e] := e' \mid \textbf{if } b \textbf{ then } u \textbf{ else } u \mid u; u \\ c ::= & x := e \mid c; c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b \textbf{ do } c \\ & \mid x := [e] \mid \langle u \rangle \mid \textbf{locked } u \mid \textbf{fork } c \mid \textbf{fail} \end{array}$$

The semantics of updates u is as follows:

$$(h, s, x := [e]) \Downarrow (h, s[x := h(s(e))]) \qquad (h, s, [e] := e') \Downarrow (h[s(e) := s(e')], s)$$

$$\frac{s(b) \qquad (h, s, u) \Downarrow (h', s')}{(h, s, \textbf{if } b \textbf{ then } u \textbf{ else } u') \Downarrow (h', s')} \qquad \frac{\neg s(b) \qquad (h, s, u') \Downarrow (h', s')}{(h, s, \textbf{if } b \textbf{ then } u \textbf{ else } u') \Downarrow (h', s')}$$

$$\frac{(h, s, u) \Downarrow (h', s') \qquad (h', s', u') \Downarrow (h'', s'')}{(h, s, u; u') \Downarrow (h'', s'')}$$

We define $[\![u]\!](s)(h) = h' \Leftrightarrow \exists s'.\ (h, s, u) \Downarrow (h', s')$.

The small-step relation $\rightsquigarrow$ on machine configurations is defined in Figure 1.

6.2 Proof System

In this logic, we support multiple TSO regions, and we allow abstract states to talk about TSO regions. Let $\mathcal{A}$ be a set of *abstract state space names*. For each $A \in \mathcal{A}$, let $\mathsf{St}(A)$ be a set of *abstract state names*, with an abstract reachability order $\preceq_A$ defined on it.

Let $\mathcal{R}$ be a set of *TSO region names*. An *abstract superstate* $\tilde{\alpha} \in \tilde{A}$ is a finite partial function from $\mathcal{R}$ to $\bigcup_{A \in \mathcal{A}} \mathsf{St}(A)$. It specifies the set of allocated TSO region names, and their current abstract state. We define $\tilde{\alpha} \preceq \tilde{\alpha}'$ as $\forall (r, \alpha) \in \tilde{\alpha}.\ \exists \alpha' \succeq \alpha.\ (r, \alpha') \in \tilde{\alpha}'$. Let sat_A be a function from $\mathsf{St}(A)$ to *semantic assertions*, i.e. sets of pairs of fractional heaps and abstract superstates. We require that all semantic assertions X be *upward-closed*: $(H, \tilde{\alpha}) \in X \land \tilde{\alpha}' \succeq \tilde{\alpha} \Rightarrow (H, \tilde{\alpha}') \in X$.

The syntax of assertions P is separation logic with fractional permissions, plus the $\mathsf{tso}_r^A(\alpha)$ assertion which states that the TSO region r has been allocated with abstract state space A and its abstract state is at least α. tso assertions may appear only in positive positions. Assertion expressions E are like program expressions e, except that they may mention logical variables X.

$$\begin{array}{l} E ::= z \mid x \mid X \mid E + E \mid E - E \\ P ::= E = E \mid E < E \mid E \overset{\pi}{\mapsto} E \mid \exists X.\ P \mid P \Rightarrow P \mid P * P \mid \textbf{emp} \mid \mathsf{tso}_r^A(\alpha) \end{array}$$

Assign
$$(h, (\overline{\delta}, s, x := e; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s[x := s(e)], \kappa) \cdot \Theta)$$

Seq
$$(h, (\overline{\delta}, s, (c; c'); \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c; c'; \kappa) \cdot \Theta)$$

IfTrue
$$\frac{s(b)}{(h, (\overline{\delta}, s, \textbf{if } b \textbf{ then } c \textbf{ else } c'; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c; \kappa) \cdot \Theta)}$$

IfFalse
$$\frac{\neg s(b)}{(h, (\overline{\delta}, s, \textbf{if } b \textbf{ then } c \textbf{ else } c'; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c'; \kappa) \cdot \Theta)}$$

WhileTrue
$$\frac{s(b)}{(h, (\overline{\delta}, s, \textbf{while } b \textbf{ do } c; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c; \textbf{while } b \textbf{ do } c; \kappa) \cdot \Theta)}$$

WhileFalse
$$\frac{\neg s(b)}{(h, (\overline{\delta}, s, \textbf{while } b \textbf{ do } c; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, \kappa) \cdot \Theta)}$$

Read
$$(h, (\overline{\delta}, s, x := [e]; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s[x := \overline{\delta}(h)(s(e))], \kappa) \cdot \Theta)$$

Enqueue
$$(h, (\overline{\delta}, s, \langle u \rangle; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta} \cdot [\![u]\!](s), s, \kappa) \cdot \Theta)$$

Locked
$$\frac{(h, s, u) \Downarrow (h', s')}{(h, (\epsilon, s, \textbf{locked } u; \kappa) \cdot \Theta) \rightsquigarrow (h', (\epsilon, s', \kappa) \cdot \Theta)}$$

Fork
$$(h, (\epsilon, s, \textbf{fork } c; \kappa) \cdot \Theta) \rightsquigarrow (h, (\epsilon, s, \kappa) \cdot (\epsilon, s, c; \textbf{done}) \cdot \Theta)$$

Dequeue
$$(h, (\delta \cdot \overline{\delta}, s, \kappa) \cdot \Theta) \rightsquigarrow (\delta(h), (\overline{\delta}, s, \kappa) \cdot \Theta)$$

Figure 1: Operational semantics of the realistic programming language

The proof rules are the standard rules of separation logic (where heap mutation is encoded as a simple update $\langle [e] := e' \rangle$), except that we add some rules and we remove some rules.

We add the following additional rules for updates and reads; these are useful for the case where insufficient permission is available locally, so a TSO space must be accessed.

Update

$$\frac{\begin{array}{c}\forall \alpha' \succeq \alpha.\ f(\alpha) \succeq \alpha \qquad \forall \alpha' \succeq \alpha, \alpha'' \succeq \alpha'.\ f(\alpha'') \succeq f(\alpha') \\ \forall \alpha' \succeq \alpha, H, \tilde{\alpha}, s.\ H, \tilde{\alpha} \in L(s) \bullet \mathsf{sat}_A(\alpha') \Rightarrow u(H, s), \tilde{\alpha} \in L'(s) \bullet \mathsf{sat}_A(f(\alpha'))\end{array}}{\{L \land \mathsf{tso}_r^A(\alpha)\}\ \langle u \rangle\ \{L' \land \mathsf{tso}_r^A(f(\alpha))\}}$$

Read

$$\frac{\forall \alpha'' \succeq \alpha, s, H, \tilde{\alpha}.\ H, \tilde{\alpha} \in L(s) \bullet \mathsf{sat}_A(\alpha'') \Rightarrow s(e) \in \mathrm{dom}(H) \land \forall v.\ H(s(e)) = (v, \_) \Rightarrow \alpha'' \succeq \alpha'(v)}{\{L \land x = v_0 \land \mathsf{tso}_r^A(\alpha)\}\ x := [e]\ \{L[v_0/x] \land \mathsf{tso}_r^A(\alpha'(x))\}}$$

We remove the disjunction rule and, correspondingly, the rule for existential quantification. However, we include restricted versions of these rules that allow case splitting on the value of a local variable.

6.3 Unsoundness of the disjunction rule

Including these rules would be unsound, since they would allow associating different abstract operations and different local postconditions with an update and different lower bounds with a read operation depending on the value of a ghost variable. That would allow one to prove that the program $\langle [30] := 2 \rangle;$ **fork** $(\langle [10] := 1 \rangle; r_1 := [20]; \langle [30] := r_1 \rangle); \langle [20] := 1 \rangle; r_2 := [10]; r_3 := [30];$ **assert** $\neg(r_3 = 0 \land r_2 = 0)$ (an encoding of the classic $((x := 1; r_1 := y) \mid\mid (y := 1; r_2 := x));$ **assert** $\neg(r_1 = 0 \land r_2 = 0)$ program) is safe, which is false. Indeed, introduce a ghost variable [40] owned by the forked thread that records the value of [20] at the time of the update of [10], and a ghost variable [50] owned by the main thread that records the value of [10] at the time of the update of [20], both initially 2. We have two abstract states: a state α_1 that states the invariant $([40] \neq 2 \Rightarrow [10] = 1) \land ([50] \neq 2 \Rightarrow [20] = 1) \land \neg([40] = 0 \land [50] = 0) \land ([30] = 0 \Rightarrow [40] = 0)$, and a state α_2 that is absurd, with $\alpha_1 \preceq \alpha_2$. For the read of [20], we pick the lower bound α_2 if $[40] = 1$ and the result of the read operation is 0, and α_1 otherwise. Therefore, after the read operation we have that $r_1 = 0 \Rightarrow [40] = 0$, which allows us to prove that the update of [30] preserves the invariant. Similarly, in the main thread, for the read of [10], we pick the lower bound α_2 if $[50] = 1$ and the result of the read operation is 0, and α_1 otherwise. Therefore, after the read operation we have that $r_2 = 0 \Rightarrow [50] = 0$. Finally, for the read of [30], we pick the lower bound α_2 if $r_2 = 0$ and the result of the read operation is 0, and α_1 otherwise.

This shows that it is unsound to allow the lower bound of a read operation to depend on ghost variables. Similarly, it is unsound to allow the abstract update or the local postcondition of an update to depend on ghost variables. To show this, introduce the additional abstract states $\alpha_3 = (\alpha_1 \land [40] = 0)$ and $\alpha_4 = (\alpha_1 \land [50] = 0)$, with $\alpha_1 \preceq \alpha_3 \preceq \alpha_2$ and $\alpha_1 \preceq \alpha_4 \preceq \alpha_2$. The abstract update for the update of [10] and [20] remains the identity function. However, after the update of [10], insert a no-op update that updates the abstract state to α_3 if $[40] = 0$ and α_1 otherwise. For the read of [20], pick lower bound α_3

for result 0, and α_1 otherwise. Similarly, after the update of [20], insert a no-op update that updates the abstract state to α_4 if $[50] = 0$ and α_1 otherwise. For the read of [10], pick lower bound α_4 if the result is zero, and α_1 otherwise. For the write of [30], the case of $[40] = 1 \wedge r_1 = 0$ is contradictory and therefore can be ignored. For the read of [30], pick lower bound α_2 if $r_2 = 0 \wedge r_3 = 0$, and α_1 otherwise.

The restricted rules that allow case splitting on a local variable do not suffer from this issue since local variables do not depend on ghost variables. (To understand this, note that the only way for information to flow from the heap to the store is through read operations, and read operations read only real variables.)

6.4 Soundness

First, an auxiliary definition: a fractional heap H and abstract superstate $\tilde{\alpha}'$ satisfy an abstract superstate $\tilde{\alpha}$ if they satisfy the separating conjunction of the abstract states of the allocated regions.

$$H, \tilde{\alpha}' \in \tilde{\alpha} \Leftrightarrow H, \tilde{\alpha}' \in \Pi_{r \in \mathrm{dom}(\tilde{\alpha})}.\ \mathsf{sat}(\tilde{\alpha}(r))$$

We prove soundness via the intermediary of a notion of validity of a machine configuration, defined analogously to Sec 5 as follows:

$$\begin{array}{l}
\mathsf{valid\_cfg}((h, \Theta_{1..n})) \Leftrightarrow \exists \tilde{\alpha}, L_{1..n}.\ (h, \tilde{\alpha}) \in \tilde{\alpha} \bullet \Pi_i L_i \wedge \mathsf{valid\_tcfg}(\tilde{\alpha}, L_i, \Theta_i) \\
\mathsf{valid\_tcfg}(\tilde{\alpha}, L, \delta \cdot \overline{\delta}, s, \kappa) = \\
\quad \exists f : \tilde{A} \rightarrow \tilde{A}, L'. \\
\quad\quad (\forall \tilde{\alpha}' \succeq \tilde{\alpha}.\ f(\tilde{\alpha}') \succeq \tilde{\alpha}') \wedge \\
\quad\quad (\forall \tilde{\alpha}' \succeq \tilde{\alpha}, \tilde{\alpha}'' \succeq \tilde{\alpha}'.\ f(\tilde{\alpha}'') \succeq f(\tilde{\alpha}')) \wedge \\
\quad\quad (\forall \tilde{\alpha}' \succeq \tilde{\alpha}, H.\ (H, \tilde{\alpha}') \in \tilde{\alpha}' \bullet L \Rightarrow (\delta(H), f(\tilde{\alpha}')) \in f(\tilde{\alpha}') \bullet L') \wedge \\
\quad\quad \mathsf{valid\_updates}(f(\tilde{\alpha}), L', \overline{\delta}, s, \kappa) \\
\mathsf{valid\_tcfg}(\tilde{\alpha}, L, \epsilon, s, \langle u \rangle; \kappa) = \mathsf{valid\_tcfg}(\tilde{\alpha}, L, [\![u]\!](s), s, \kappa) \\
\mathsf{valid\_tcfg}(\tilde{\alpha}, L, \epsilon, s, x := [e]; \kappa) = \\
\quad \forall v. \\
\quad\quad (\forall \tilde{\alpha}'' \succeq \tilde{\alpha}, H.\ (\tilde{\alpha}'', H) \in \tilde{\alpha}'' \bullet L \Rightarrow \\
\quad\quad\quad s(e) \in \mathrm{dom}(H) \wedge (H(s(e)) = (v, \_) \Rightarrow \mathsf{valid\_tcfg}(\tilde{\alpha}, L, \epsilon, s[x := v], \kappa))) \vee \\
\quad\quad \exists \tilde{\alpha}'. \\
\quad\quad\quad (\forall \tilde{\alpha}'' \succeq \tilde{\alpha}, H.\ (H, \tilde{\alpha}'') \in \tilde{\alpha}'' \bullet L \Rightarrow s(e) \in \mathrm{dom}(H) \wedge (H(s(e)) = (v, \_) \Rightarrow \tilde{\alpha}'' \succeq \tilde{\alpha}')) \wedge \\
\quad\quad\quad \mathsf{valid\_tcfg}(\tilde{\alpha}', L, \epsilon, s[x := v], \kappa)
\end{array}$$

This notion of validity is sound. The proofs are analogous to those of Section 5.

Lemma 4. *If* $\gamma \rightsquigarrow \gamma'$ *and* $\mathsf{valid\_cfg}(\gamma)$ *then* $\mathsf{valid\_cfg}(\gamma')$.

Lemma 5. *If* $\mathsf{valid\_cfg}(\gamma)$ *then* $\gamma \not\rightsquigarrow^* \mathit{Fail}$.

We prove a correspondence between the Hoare rules and the notion of validity.

Lemma 6. *If* $\vdash \{P\}\ c\ \{Q\}$ *and* $\forall s'.\ \exists \tilde{\alpha}'.\ (\forall (H, \tilde{\alpha}'') \in [\![Q]\!]_{I,s'}.\ \tilde{\alpha}'' \succeq \tilde{\alpha}') \wedge \mathsf{valid}(\tilde{\alpha}', [\![Q]\!]_{I,s'}, s', \kappa)$ *and* $\forall (H, \tilde{\alpha}') \in L.\ \tilde{\alpha}' \succeq \tilde{\alpha} \Rightarrow (H, \tilde{\alpha}') \in [\![P]\!]_{I,s}$ *then* $\mathsf{valid}(\tilde{\alpha}, L, s, c; \kappa)$.

Proof. By induction on the derivation of the Hoare triple.

- **Case** ASSIGN. OK.
- **Case** MUTATE. Take $f = \lambda\tilde{\alpha}.\ \tilde{\alpha}$.
- **Case** LOOKUP. OK.
- **Case** UPDATE. OK.
- **Case** READ. OK.
- **Case** SEQ. OK.
- **Case** CONSEQ. OK.

□

We can now prove the soundness of our Hoare logic with respect to the operational semantics of the programming language.

Theorem 3. *If* $\{\mathbf{emp}\}\ c\ \{\mathbf{true}\}$ *then* $(\emptyset, \{\!\{(\epsilon, \emptyset, c; \mathbf{done})\}\!\}) \not\rightsquigarrow^* \mathit{Fail}$.

Proof. Apply Lemma 6 with $\tilde{\alpha} = \tilde{\alpha}' = \emptyset$ and $L = \{(\emptyset, \_)\}$ to obtain $\mathsf{valid}(\emptyset, L, \emptyset, c; \mathbf{done})$ and therefore $\mathsf{valid\_cfg}(\emptyset, \{\!\{(\epsilon, \emptyset, c; \mathbf{done})\}\!\})$. We obtain the goal by Lemma 5. □

7 Tool support

We developed a preliminary encoding of the proof rules for TSO memory accesses of the preceding section into our VeriFast sound modular static verification tool for C programs, and we checked versions of the examples (VM, lock, producer-consumer) in the C programming language using this encoding. These examples are included with the latest VeriFast distribution in the directory `examples/tso`.

In the development of our encoding, we needed to take special care to make sure that the abstract updates associated with TSO updates and the abstract lower bounds associated with TSO reads do not depend on ghost variables. Indeed, the naive approach of specifying the abstract updates and abstract lower bounds as ghost arguments of the TSO operations is unsound, since VeriFast allows ghost arguments to depend on ghost variables.

In our current encoding, we work around this issue by requiring the list of all abstract updates and abstract lower bounds to be used by operations on a given TSO space to be specified when the TSO space is created. The specific abstract update or lower bound for a particular operation is then selected by passing an index into this list as a non-ghost argument to the TSO operation (which appears in the program as a C function call). Furthermore, to allow the abstract updates and lower bounds to depend on the (non-ghost) state of the thread, we allow a variable number of additional non-ghost arguments to be passed to the TSO operations. These are passed on as extra arguments to the abstract updates and lower bounds.

This encoding is sound but it has the downside that it requires modifications to the C program: extra arguments to the TSO operations, and extra variables to track the thread state.

A better approach which we envision for future work is to extend VeriFast with support for additional ghost-levels of code and variables, beyond the level of reality (the lowest ghost-level) and the single existing ghost- level. Information flow from higher to lower ghost-levels would be disallowed. For the TSO encoding, we would use three ghost-levels: the real level, the semi-ghost level, and the full ghost level. Full ghost variables can be modified as part of TSO updates; abstract updates and lower bounds are specified as semi-ghost arguments.

8 Conclusion

We presented an approach for the modular formal verification of programs that use memory accesses with x86-TSO semantics.

A comparison with related work is future work.

Acknowledgments

The author thanks Ernie Cohen for very helpful comments and discussions. This work was partially funded by EU FP7 FET-Open project ADVENT (grant number 308830).