A Rely-Guarantee proof system for x86-TSO

Tom Ridge

University of Leicester

Abstract. Current multiprocessors provide weak or relaxed memory models. Existing program logics assume sequential consistency, and are therefore typically unsound for weak memory. We introduce a novel Rely-Guarantee style proof system for reasoning about x86 assembly programs running against the weak x86-TSO memory model. Interesting features of the logic include processor assertions which can refer to the local state of other processors (including their program counters), and a syntactic operation of closing an assertion under write buffer interference. We use the expressivity of the proof system to construct a new correctness proof for an x86-TSO version of Simpson's four slot algorithm. Mechanization in the Hol theorem prover provides a flexible tool to support semi-automated verification.

1 Introduction

Multiprocessors are now widespread, but real multiprocessors provide subtle relaxed (or weak) memory models. Typically sequential consistency (SC) can be recovered by appropriate programming disciplines eg the use of locks to guard access to shared memory. However, there are several areas where the use of locks is either not possible, or would impose unacceptably high performance costs. For example, operating system lock implementations cannot assume that locks are already provided, and non-blocking synchronization techniques avoid the use of locks to provide good performance and strong progress guarantees. However, these programs are directly exposed to the weak memory models of the underlying processors, and consequently there is often considerable doubt about whether they are correct [Lin99]. Unfortunately existing program logics are typically no longer sound in this setting.

Our main contribution in Sect. 5 is a proof system for processors executing x86 assembly code with the x86-TSO memory model, proved sound with respect to the operational semantics. In Sect. 6 we show that the system is pragmatically useful by using it to give a novel proof of Simpson's four slot algorithm [Sim90]. The Hol mechanization[1] is a formal version of this paper, including complete definitions, formal proof rules, formal soundness proofs, the example application to Simpson's algorithm, and a mechanized proof environment to tackle further examples. We now discuss some interesting features of our program logic.

A rigorous semantics for the relaxed x86-TSO memory model has been defined in higher-order logic and mechanized in the Hol theorem prover [OSS09],

[1] Available online at http://www.cs.le.ac.uk/people/tr61/vstte2010

see Sect. 3. We extend this model with an operational semantics for x86 assembly code in Sect. 4. For x86-TSO, every processor is connected to main memory by a FIFO write buffer modelled as a list of (address, value) pairs. Each write buffer process repeatedly removes a write from the head of the queue, and commits the write to main memory. A processor indirects via its write buffer: a read returns the value of the last buffered write to the address, if any, otherwise the value of the address in main memory, as usual. *We treat write buffers as active processes.* This is not straightforward: in traditional models, processes cannot write to or read from each other's local state, whereas here a processor affects the local state of its write buffer whenever it tries to write to a memory address. For x86-TSO, a processor's write buffer state is writable by that processor, but inaccessible to other processors. The need for *private state, that is shared between two related processes*, is built into our proof system. Write buffers also affect the semantics of assertions. Traditionally, the validity of a process assertion should not be affected by the behaviour of other processes. We require that the *validity is unaffected by the behaviour of the write buffer processes.* Writing syntactic assertions that satisfy this constraint is difficult, so we introduce a *syntactic operation of "closing an assertion under write buffer interference".* This is a key step towards making x86-TSO verification tractable.

The main challenge of low-level x86 assembly code is the *non-atomic nature of individual instructions.* For example, the $\texttt{mov}_{\mathsf{iload}}$ `eax`, `ebx` (indirect load) instruction loads the register `eax` with the value of the memory address pointed to by `ebx`. This accounts for three separate memory/register read/writes. Interference from other processors can occur between each of these steps. The problem here is that individual instructions are *not* atomic. We do not give a general solution, but try hard to make the proof rules as simple as possible.

In traditional proof systems the notion of state is restricted: program counters (or equivalent) are not part of the state, and process assertions cannot refer to the local state of other processes. *We lift both these restrictions*, thereby dramatically increasing the expressivity of the system. The new elegant proof of Simpson's algorithm we present uses *processor assertions that refer to the private state of other processors.* Moreover, it seems very natural to talk about processes executing different regions of their code, and this essentially involves *assertions about processes' program counters.* A specific motivation for x86-TSO is that a processor assertion will often refer to the state of the processor's write buffer. Unfortunately this increases the complexity of the proof rules. For example, the familiar assertion $\{P\}$ `nop` $\{P\}$ is no longer valid eg if P is "the next instruction is `nop`", and `nop` is followed by a non-`nop` instruction. In order to recover the familiar $\{P\}$ `nop` $\{P\}$ rule, we require that the assertion P is invariant under changes to the processor's current instruction. Fortunately these side conditions are trivial in practice.

Notation Formal definitions have been lightly edited before inclusion here. We use the following notation: function update ($f \oplus (x, y)$); list concatenation ($xs +\!\!+\ ys$); head and tail of a list ($\mathsf{HD}\ xs$ and $\mathsf{TL}\ xs$); records with fields having values ($\langle\!|\ \mathsf{fld} = v;\ \ldots\ |\!\rangle$); record update of a field with a value

(r with { fld $= v$ }); domain of a function (DOM f); image of a function or relation on a set (IMAGE f S); restriction of the domain of a function or relation to a set ($f|_S$); function application (f pid or f_{pid}).

2 Preview of the proof system

In this section we give the syntax and semantics of the main proof system judgement informally. The familiar Hoare triple $\vdash \{P\}\ c\ \{Q\}$ is valid iff starting from a state satisfying P, execution of the x86 assembly instruction c ends in a state satisfying Q [Flo67,Hoa69]. To this we add the standard Rely-Guarantee relations (R, G) to give the judgement $(R, G) \vdash \{P\}\ c\ \{Q\}$: We now consider the execution of c interleaved with steps of other processes, which we can *assume* are approximated by the set of transitions R (Rely assumption). Dually we must *prove* that G approximates c steps (Guarantee commitment) [Jon81]. The judgement is valid iff starting from P, executing c steps interleaved with R steps, execution ends in Q and furthermore every c step is contained in G [2]. Finally we must address what happens when execution jumps to some other address. We include in the judgement a component J such that J.invs is a partial map from code points, to invariants that must hold when execution reaches that point [CH72]. In the case that c terminates by jumping to a code point lbl, the final state must satisfy J.invs lbl rather than Q. To this we add the processor pid that is executing c and the code point ma of the current instruction (both of which can typically be ignored) to get $pid,\ (R, G),\ J,\ ma \vdash \{P\}\ c\ \{Q\}$. If the judgement is valid, we write $pid,\ (R, G),\ J,\ ma \models \{P\}\ c\ \{Q\}$. Some example judgements are:

- $pid,\ (\{\},\ \{(s, s') \mid \mathsf{T}\}),\ J,\ ma \vdash \{\mathsf{T}\}\ \mathtt{nop}\ \{\mathsf{ma} = ma + 1;\ \mathsf{ci} = [ma + 1]\}$, where the precondition $\{\mathsf{T}\}$ is unconstrained, and the post-condition states that the current code point is $ma + 1$ and the current instruction is whatever instruction was stored in memory at address $ma + 1$.
- $pid,\ (\{\},\ \{(s, s') \mid \mathsf{T}\}),\ J,\ ma \vdash \{J.\mathsf{invs}\ lbl\}\ \mathsf{jump}\ lbl\ \{\bot\}$, where the invariant that must hold after the jump is already established in the pre-condition.

An example proof rule concerns the instruction $\mathtt{mov}_{\mathsf{ri}}(r, n)$, which sets local register r to the value n:

$$\frac{\begin{array}{c}\mathsf{wf}\ (pid,\ (R, G),\ J,\ ma \vdash \{P\}\ \mathtt{mov}_{\mathsf{ri}}(r,\ n)\ \{Q\}) \\ \mathsf{nop\_conditions}\ pid\ P\ Q\ G \\ f = \mathsf{update\_f}\ pid\ (\lambda\ ll.\ ll\ \mathsf{with}\ \{\ \mathsf{l} = ll.\mathsf{l} \oplus (r, n);\ \mathsf{ci} = \mathtt{nop}\ \}) \\ \mathsf{IMAGE}\ f\ P \subseteq Q \qquad f\ |_P \subseteq G\end{array}}{pid,\ (R, G),\ J,\ ma \models \{P\}\ \mathtt{mov}_{\mathsf{ri}}(r,\ n)\ \{Q\}}\ \mathtt{MOV}_{\mathrm{RI}}$$

The first condition checks that the judgement is well-formed, which includes the usual Rely-Guarantee requirement that P and Q are closed under R [3]. A formal

[2] A key point for x86-TSO (but not in the semantics presented in this section) is that R includes *at least* all those steps that can be taken by write buffers. Thus, to prove a judgement valid using our proof system, it is necessary to take into account the behaviour of all write buffers.

[3] A set P is closed under a relation R iff for all s in P, for all s', if (s, s') in R then s' is also in P.

definition of judgement well-formedness will be given shortly. In addition there is a technical side-condition related to `nop` transitions, which can be safely ignored for now. The operational semantics for $\texttt{mov}_{\mathsf{ri}}(r, n)$ simply updates the local state $ll.\mathsf{l}$ of processor pid at register r with the value n whilst also updating the current instruction ci to `nop`. This is captured by the update function f. The judgement is valid iff starting from a state $s \in P$, the update function results in a state $f\ s \in Q$ (ie $\mathsf{IMAGE}\ f\ P \subseteq Q$), and in addition the update is allowed by the guarantee G (ie $f\ |_P \subseteq G$, where the function f is considered as a set of pairs).

3 The x86-TSO memory model

We briefly review the x86-TSO memory model [OSS09], which usefully abstracts from the details of x86 assembly instructions. The model consists of processors connected via write buffers to a single shared main memory. The model also includes details of the per-processor local registers. *Individual* x86 instructions can be locked (and so execute atomically) which is captured by a lock value L, indicating which processor if any currently holds the lock. The states of the x86-TSO machine are records with the following fields:

```
machine_state = {
  R : proc → reg → value option;     // per processor registers
  M : address → value option;        // main memory
  B : proc → (address × value) list; // per processor write buffers
  L : proc option                    // which processor holds the lock, if any
}
```

The behaviour of the system is described by the labelled transition relation $s \xrightarrow{lbl} s'$ in Fig. 1. The datatype of labels is label = Tau | Evt of proc × action | `lock` of proc | `unlock` of proc where an action is either a memory barrier or a read or write to a register or memory address. The predicate not_blocked $s\ p$ holds if the lock is owned by processor p, or if the lock is not held by any processor. The predicate no_pending $xs\ a$ checks that there are no writes to address a in write buffer xs.

4 x86 assembly code

We rephrase the model of the previous section and extend it with a model of x86 assembly code. The syntax of assembly instructions is expressed as a datatype:

```
instruction =                          | mov_iload of reg_name × reg_name
  | nop                                | jump of flag_condition × code_point
  | mov_ri of reg_name × value         | lock of instruction
  | mov_rr of reg_name × reg_name      | Barrier Mfence
  | mov_rm of reg_name × data_address  | ...
```

The state of the system, type S, is a record with a field g giving the contents of the shared memory, a field f giving the local state for each processor, and a field lck giving the processor that holds the lock, if any. The local state LL for each processor consists of some code, the address of the current instruction ma,

Read from memory

$$\frac{\mathsf{not\_blocked}\ s\ p \qquad s.M\ a = \mathsf{SOME}\ v \qquad \mathsf{no\_pending}\ (s.B\ p)\ a}{s \xrightarrow{\mathsf{Evt}\ p\ (\mathsf{Access}\ R\ (\mathsf{Location\_mem}\ a)\ v)} s}$$

Read from write buffer

$$\frac{\mathsf{not\_blocked}\ s\ p \qquad s.B\ p = b1 +\!\!+ [(a,v)] +\!\!+ b2 \qquad \mathsf{no\_pending}\ b1\ a}{s \xrightarrow{\mathsf{Evt}\ p\ (\mathsf{Access}\ R\ (\mathsf{Location\_mem}\ a)\ v)} s}$$

Read from register

$$\frac{s.R\ p\ r = \mathsf{SOME}\ v}{s \xrightarrow{\mathsf{Evt}\ p\ (\mathsf{Access}\ R\ (\mathsf{Location\_reg}\ p\ r)\ v)} s}$$

Write to write buffer

$$s \xrightarrow{\mathsf{Evt}\ p\ (\mathsf{Access}\ W\ (\mathsf{Location\_mem}\ a)\ v)} s\ \mathsf{with}\ \{\ B = s.B \oplus (p, [(a,v)] +\!\!+ (s.B\ p))\ \}$$

Write from write buffer to memory

$$\frac{\mathsf{not\_blocked}\ s\ p \qquad s.B\ p = b +\!\!+ [(a,v)]}{s \xrightarrow{\mathsf{Tau}} s\ \mathsf{with}\ \{\ M = s.M \oplus (a, \mathsf{SOME}\ v);\ B = s.B \oplus (p,b)\ \}}$$

Write to register

$$s \xrightarrow{\mathsf{Evt}\ p\ (\mathsf{Access}\ W\ (\mathsf{Location\_reg}\ p\ r)\ v)} s\ \mathsf{with}\ \{\ R = s.R \oplus (p,\ (s.R\ p) \oplus (r,\ \mathsf{SOME}\ v))\ \}$$

Barrier

$$\frac{s.B\ p = []}{s \xrightarrow{\mathsf{Evt}\ p\ (\mathsf{Barrier}\ \mathsf{Mfence})} s}$$

Lock

$$\frac{s.L = \mathsf{NONE} \qquad s.B\ p = []}{s \xrightarrow{\mathtt{lock}\ p} s\ \mathsf{with}\ \{\ L = \mathsf{SOME}\ p\ \}}$$

Unlock

$$\frac{s.L = \mathsf{SOME}\ p \qquad s.B\ p = []}{s \xrightarrow{\mathtt{unlock}\ p} s\ \mathsf{with}\ \{\ L = \mathsf{NONE}\ \}}$$

Fig. 1. The x86-TSO machine behaviour [OSS09]

the state of the current instruction ci, the values of the processor-local registers l, and the pending writes in the write buffer w.

$$\boxed{\begin{array}{ll}
S = \{ & LL = \{ \\
\quad \mathsf{g} :\ \mathsf{data\_address} \rightarrow_{\mathit{fin}} \mathsf{value}; & \quad \mathsf{code}\ :\ \mathsf{code\_point} \rightarrow_{\mathit{fin}} \mathsf{instruction}; \\
\quad \mathsf{f} :\ \mathsf{proc} \rightarrow_{\mathit{fin}} LL; & \quad \mathsf{ma}\ :\ \mathsf{code\_point}; \\
\quad \mathsf{lck} :\ \mathsf{proc\ option} & \quad \mathsf{ci}\ :\ \mathsf{instruction}; \\
\} & \quad \mathsf{l} :\ \mathsf{reg\_name} \rightarrow_{\mathit{fin}} \mathsf{value}; \\
 & \quad \mathsf{w} :\ (\mathsf{data\_address} \times \mathsf{value})\ \mathsf{list} \\
 & \}
\end{array}}$$

The proc_view function gives a processor's view of memory, taking into account pending writes in the write buffer:

$$\boxed{\mathsf{proc\_view}\ \mathit{pid}\ s = \mathsf{list.FOLDR}\ (\lambda\ (a,v).\ \lambda\ g.\ g \oplus (a,v))\ s.\mathsf{g}\ (s.\mathsf{f}\ \mathit{pid}).\mathsf{w}}$$

The semantics is then expressed as a (small-step) state transition relation TransP $\mathit{pid}\ (s, s')$. This uses several auxiliary relations. The first, PreTransP, gives the basic semantics of commands.

```
PreTransP $pid$ $s$ =
  case ($s$.f $pid$).ci of
  nop $\rightarrow$ failwith "PreTransP : nop"
  || $\mathtt{mov}_{\mathsf{ri}}(r,n)$ $\rightarrow$ update_f $pid$ ($\lambda$ $ll$. $ll$ with { l = $ll$.l $\oplus$ $(r,n)$; ci = nop }) $s$
  || $\mathtt{mov}_{\mathsf{rm}}(r,a)$ $\rightarrow$ update_f $pid$ ($\lambda$ $ll$. $ll$ with { ci = $\mathtt{mov}_{\mathsf{ri}}(r,$ proc_view $pid$ $s$ $a)$ }) $s$
...
```

Note that most instructions (including locked instructions) are eventually rewritten to nop. The evaluation of a nop instruction involves getting the next instruction at address ma $+ 1$ and releasing the lock if it is taken.

```
XnopTrans $pid$ $(s,s')$ =
  let $ll$ = $s$.f $pid$ in
  case $ll$.ci of
  nop $\rightarrow$ (
      let $ma'$ = $ll$.ma + 1 in
      if $ma' \notin$ DOM $ll$.code then $\bot$ else
      let $s1$ = update_f $pid$ ($\lambda$ $ll$. $ll$ with { ma = $ma'$; ci = ($ll$.code $ma'$) }) $s$ in
      let $s2$ = unset_lock $s1$ in
      $s' = s2$)
  || _ $\rightarrow$ $\bot$
```

The lock and jump transitions are handled similarly. TransP is then

```
TransP $pid$ $(s,s')$ =
  $pid \in$ DOM $s$.f $\wedge$ not_blocked $pid$ $s$ $\wedge$ let $ll$ = $s$.f $pid$ in
  case $ll$.ci of
  nop $\rightarrow$ (XnopTrans $pid$ $(s,s')$)
  || lock $c$ $\rightarrow$ (XlockTrans $pid$ $(s,s')$)
  || jump$(c,n)$ $\rightarrow$ (XjumpTrans $pid$ $(s,s')$)
  || _ $\rightarrow$ ($s'$ = PreTransP $pid$ $s$)
```

The write buffer for processor pid simply takes the first pending write of value v to address a and updates the global memory g.

```
TransWb $pid$ $(s,s')$ =
  let $ll$ = $s$.f $pid$ in
  let $(a,v)$ = HD $ll$.w in
  let $s1$ = $s$ with { g = $s$.g $\oplus$ $(a,v)$ } in
  let $s2$ = update_f $pid$ ($\lambda$ $ll$. $ll$ with { w = TL $ll$.w }) $s1$ in
  $pid \in$ DOM $s$.f $\wedge$ $ll$.w $\neq$ [] $\wedge$ ($s' = s2$)
```

The transitions of the system are simply the union of the individual processor and write buffer transitions, TransS $= \bigcup_{pid}\{$TransP $pid \cup$ TransWb $pid\}$.

For example, a simple instruction such as $\mathtt{mov}_{\mathsf{ri}}(r,n)$ executes in two steps, the first PreTransP transition updates the local state, and changes the current instruction to nop. The second XnopTrans transition gets the next instruction from memory, then updates the current instruction and current address. In the concurrent setting, these steps are interleaved with steps of write buffers and other processors. More complicated instructions take more than two steps to execute, and each step may involve accessing an address in memory or a local register. This lack of atomicity impacts considerably on the proof system.

5 Rely-Guarantee proof system

In this section we give our main contribution, a Rely-Guarantee proof system for x86 assembly code with the x86-TSO memory model. In Sect. 2 we gave the syntax and semantics of our judgement, and discussed the $\texttt{MOV}_{\text{RI}}$ rule. In Fig. 2 we give selected rules covering further x86 instructions and logical aspects such as weakening.

Judgement well-formedness and soundness The non-logical rules use a judgement well-formedness condition wf, and typically also include a $\mathsf{nop\_conditions}$ side condition. The main aim of these conditions is to reduce the complexity of rules, which results from the liberal notion of state and the non-atomic nature of individual instructions, by making assumptions about P, Q, R, G. We motivate these conditions by discussing in more detail the soundness proof for rule $\texttt{MOV}_{\text{RI}}$ from Sect. 2. The execution of $\texttt{mov}_{\mathsf{ri}}(r,n)$ is interleaved with R-steps as follows:

$$s_0 \xrightarrow{R^*} s_1 \xrightarrow{\mathsf{PreTransP}} s_2 \xrightarrow{R^*} s_3 \xrightarrow{\mathsf{XnopTrans}} s_4 \xrightarrow{R^*} s_5$$

We are given that $s_0 \in P$, and we need to show $s_5 \in Q$. Our well-formedness assumption gives that P and Q are closed under R (this is a standard assumption), so it suffices to assume $s_1 \in P$, and show $s_4 \in Q$. Formally, wf is defined as follows:

$$\boxed{\begin{array}{l}\mathsf{wf}\ j = \mathsf{case}\ j\ \mathsf{of}\ \mathit{pid}\ (R,G)\ J\ \mathit{ma} \vdash P\ c\ Q \to \\ \quad \mathsf{wf\_R}\ \mathit{pid}\ R \wedge \mathsf{wf\_G}\ \mathit{pid}\ G \wedge (\mathsf{closed}\ P\ R) \wedge (\mathsf{closed}\ Q\ R)\end{array}}$$

The judgements $\mathsf{wf\_R}$, $\mathsf{wf\_G}$ are technical conditions that assert eg that the rely for a processor preserves for values of that processor's local registers. The $\mathsf{nop\_conditions}$ in the premises of the rule require Q to be closed under $\mathsf{XnopTrans}\ \mathit{pid}$ transitions:

$$\boxed{\begin{array}{l}\mathsf{nop\_conditions}\ \mathit{pid}\ P\ Q\ G = \\ \quad \mathsf{closed}\ Q\ (\mathsf{XnopTrans}\ \mathit{pid}) \\ \quad \wedge \ \{\ (s,s') \mid (s,s') \in \mathsf{XnopTrans}\ \mathit{pid} \wedge s' \in Q\ \} \subseteq G\end{array}}$$

so it suffices to show $s_2 \in Q$ (using again the fact that Q is closed under R). In practice, assertions for processor pid do not mention pid's program counter, and the side condition is trivial. Let $f = \mathsf{update\_f}\ \mathit{pid}\ (\lambda\ \mathit{ll}.\ \mathit{ll}\ \mathsf{with}\ \{\ \mathsf{l} = \mathit{ll}.\mathsf{l} \oplus (r,n);\ \mathsf{ci} = \texttt{nop}\ \})$. From the definition of $\mathsf{PreTransP}$, we have that $s_2 = f\ s_1$ ie $s_2 \in \mathsf{IMAGE}\ f\ P$. So the rule is sound only if $\mathsf{IMAGE}\ f\ P \subseteq Q$, one of the premises of the rule. A similar argument can be used to prove that the Guarantee commitment is satisfied only if $f|_P \subseteq G$.

Composition Our judgement concerns a single processor pid executing in some environment R which has so far been largely unconstrained. For x86-TSO, R should include the transitions of other processors and all write buffers. The move from the global view of the system with many processors and write buffers, to the local view of a single processor in environment R is handled by the COMP rule. $\hat{J}$ is a function that for each pid gives a $J = \hat{J}\ \mathit{pid}$. Well-formedness of $\hat{J}$ means that each J should provide a pre and post condition for each code point (not just jump targets). Similarly $\hat{R}$, $\hat{G}$ give rise to R and G. The first Rely-Guarantee requirement is that $\hat{R}\ \mathit{pid}$, the rely for pid, should

$$\frac{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P'\}\ c\ \{Q'\} \qquad P \subseteq P' \qquad Q' \subseteq Q}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ c\ \{Q\}}\ \textsc{Weaken}$$

$$\frac{\mathit{pid},\ (R',G'),\ J,\ \mathit{ma} \models \{P\}\ c\ \{Q\} \qquad R \subseteq R' \qquad G' \subseteq G}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ c\ \{Q\}}\ \textsc{WeakenRG}$$

$$\frac{\mathit{pid},\ (R,G),\ J',\ \mathit{ma} \models \{P\}\ c\ \{Q\} \qquad J' \subseteq J}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ c\ \{Q\}}\ \textsc{WeakenJ}$$

$$\frac{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ c\ \{Q\} \qquad \mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P'\}\ c\ \{Q'\}}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P \wedge P'\}\ c\ \{Q \wedge Q'\}}\ \textsc{Conj}$$

$$\frac{\mathsf{wf}\ (\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \vdash \{P\}\ \mathtt{nop}\ \{P\}) \qquad \mathsf{nop\_conditions}\ \mathit{pid}\ P\ P\ G}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ \mathtt{nop}\ \{P\}}\ \mathtt{NOP}$$

$$\frac{\begin{array}{c}\mathsf{wf}\ (\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \vdash \{P\}\ \mathtt{mov}_{\mathsf{rr}}(\mathit{r1},\ \mathit{r2})\ \{Q\}) \qquad s \in P \\ v = (s.\mathsf{f}\ \mathit{pid}).\mathsf{l}\ \mathit{r2} \qquad f = \mathsf{update\_f}\ \mathit{pid}\ (\lambda\ \mathit{ll}.\ \mathit{ll}\ \mathsf{with}\ \{\ \mathsf{ci} = \mathtt{mov}_{\mathsf{ri}}(\mathit{r1}, v)\ \}) \\ (s, f\ s) \in G \qquad \mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{f\ s\}\ \mathtt{mov}_{\mathsf{ri}}(\mathit{r1}, v)\ \{Q\}\end{array}}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ \mathtt{mov}_{\mathsf{rr}}(\mathit{r1},\ \mathit{r2})\ \{Q\}}\ \mathtt{MOV}_{\mathrm{RR}}$$

$$\frac{\begin{array}{c}\mathsf{wf}\ (\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \vdash \{P\}\ \mathtt{mov}_{\mathsf{rm}}(r,\ a)\ \{Q\}) \qquad s \in P \\ v = \mathsf{proc\_view}\ \mathit{pid}\ s\ a \qquad f = \mathsf{update\_f}\ \mathit{pid}\ (\lambda\ \mathit{ll}.\ \mathit{ll}\ \mathsf{with}\ \{\ \mathsf{ci} = \mathtt{mov}_{\mathsf{ri}}(r, v)\ \}) \\ (s, f\ s) \in G \qquad \mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{f\ s\}\ \mathtt{mov}_{\mathsf{ri}}(r, v)\ \{Q\}\end{array}}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ \mathtt{mov}_{\mathsf{rm}}(r, a)\ \{Q\}}\ \mathtt{MOV}_{\mathrm{RM}}$$

$$\frac{\begin{array}{c}\mathsf{wf}\ (\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \vdash \{P\}\ \mathtt{mov}_{\mathsf{iload}}(\mathit{r1},\ \mathit{r2})\ \{Q\}) \qquad s \in P \\ a = (s.\mathsf{f}\ \mathit{pid}).\mathsf{l}\ \mathit{r2} \qquad f = \mathsf{update\_f}\ \mathit{pid}\ (\lambda\ \mathit{ll}.\ \mathit{ll}\ \mathsf{with}\ \{\ \mathsf{ci} = \mathtt{mov}_{\mathsf{rm}}(\mathit{r1}, a)\ \}) \\ (s, f\ s) \in G \qquad \mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{f\ s\}\ \mathtt{mov}_{\mathsf{rm}}(\mathit{r1}, a)\ \{Q\}\end{array}}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ \mathtt{mov}_{\mathsf{iload}}(\mathit{r1},\ \mathit{r2})\ \{Q\}}\ \mathtt{MOV}_{\mathrm{ILOAD}}$$

$$\frac{\begin{array}{c}\mathsf{wf}\ (\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \vdash \{P\}\ \mathtt{jump}(\mathit{cnd},\ \mathit{ma}')\ \{Q\}) \\ \mathsf{nop\_conditions}\ \mathit{pid}\ P\ Q\ G \\ \mathsf{closed}\ (J.\mathsf{invs}\ \mathit{ma}')\ R \qquad \mathsf{IMAGE}\ \mathsf{XjumpTrans}_{\mathit{pid}}\ (P \cap \mathit{cnd}) \subseteq J.\mathsf{invs}\ \mathit{ma}' \\ \mathsf{IMAGE}\ \mathsf{XjumpTrans}_{\mathit{pid}}\ (P \cap \neg\mathit{cnd}) \subseteq Q \qquad \mathsf{XjumpTrans}_{\mathit{pid}}|_P \subseteq G\end{array}}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ \mathtt{jump}(\mathit{cnd},\ \mathit{ma}')\ \{Q\}}\ \mathtt{JUMP}$$

$$\frac{\begin{array}{c}\mathsf{wf}\ (\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \vdash \{P\}\ \mathtt{lock}(c)\ \{Q\}) \\ \mathit{pid},\ (\{\},\ G),\ J,\ \mathit{ma} \models \{P\}\ c\ \{Q\} \qquad \mathsf{XlockTrans}_{\mathit{pid}}|_P \subseteq G\end{array}}{\mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{P\}\ \mathtt{lock}(c)\ \{Q\}}\ \mathtt{LOCK}$$

$$\frac{\begin{array}{c}\mathsf{wf}\ \hat{J} \wedge \exists\ \hat{R}\ \exists\ \hat{G} \\ (\forall\ \mathit{pid}.\ \bigcup_{\mathit{pid}' \neq \mathit{pid}} (\hat{G}\ \mathit{pid}') \subseteq (\hat{R}\ \mathit{pid})) \\ \wedge\ (\forall\ \mathit{pid}.\ \bigcup_{\mathit{pid}'} (\mathsf{TransWb}\ \mathit{pid}') \subseteq (\hat{R}\ \mathit{pid})) \\ \wedge\ (\forall\ \mathit{ma}.\ \mathsf{let}\ J = \hat{J}\ \mathit{pid}\ \mathsf{in} \\ \mathsf{let}\ (R,\ G) = (\hat{R}\ \mathit{pid},\ \hat{G}\ \mathit{pid})\ \mathsf{in} \\ \mathit{pid},\ (R,G),\ J,\ \mathit{ma} \models \{J.\mathsf{invs}\ \mathit{ma}\}\ J.\mathsf{code}\ \mathit{ma}\ \{J.\mathsf{invs}\ (\mathit{ma}+1)\})\end{array}}{\models_{\hat{\jmath}}\ \hat{J}}\ \textsc{Comp}$$

Fig. 2. Selected proof rules

contain the guarantees of the other processors pid'. The second requirement is that the rely also contain all write buffer transitions. The final conjunct requires that for every instruction $c = J.\mathsf{code}\ ma$, there is a valid judgement $pid,\ (R, G),\ J,\ ma \models \{J.\mathsf{invs}\ ma\}\ c\ \{J.\mathsf{invs}\ (ma + 1)\}$. The meaning of the conclusion $\models_{\hat{\mathsf{J}}} \hat{J}$ is that, providing the system starts in a state s where the instruction executed by pid and identified by ma is such that $s \in (\hat{J}\ pid).\mathsf{invs}\ ma$, then all further invariants given by $(\hat{J}\ pid).\mathsf{invs}\ ma'$ hold whenever execution of pid reaches ma'. Thus $\models_{\hat{\mathsf{J}}} \hat{J}$ represents the conjunction of invariants, each of which are indexed by pid and ma. The invariants themselves can be arbitrary formulas in higher-order logic, over the whole system state (including program counters).

The logic in practice The Comp rule requires that the Rely relations include at least the transitions of the write buffers. A consequence is that the proof rules for instructions require assertions to be closed under write buffer transitions. For example, suppose we write a value v to an address a. We might incorrectly annotate the write as $\{\mathsf{w} = []\}a := v\{\mathsf{w} = [(a, v)]\}$, where w informally refers to the write buffer of the process. However, the assertion $\{\mathsf{w} = [(a, v)]\}$ is not closed, because at any point after the instruction executes, the write buffer could flush the write and become empty. A correct assertion, that is closed under write buffer interference, is $\{\mathsf{w} = [(a, v)] \vee \mathsf{w} = []\}$. Rather than expanding various possible states as disjuncts, a more succinct approach is to allow assertions to be explicitly closed with respect to write buffer interference. If P is an assertion $\{\dots\}$, we write the closure under R as $\{\dots\}^R$. For example, if R represents interference from the write buffer, then $\{\mathsf{w} = [(a, v)]\}^R = \{\mathsf{w} = [(a, v)] \vee \mathsf{w} = []\}$. In practice, rather than deal with the whole write buffer, we often want to refer to pending writes to a particular address. We introduce the syntax $\{a \doteq xs\}$ to mean that the contents of the write buffer, filtered to address a (and then projected onto the written values), is xs. For example, if $a \neq b$, then a write buffer $[(b, 0); (a, 1); (b, 2); (a, 3)]$ satisfies the assertion $\{a \doteq [1, 3]\}$.

A key point is that our assertions are higher-order logic predicates over the entire system state S, and can therefore be almost arbitrarily complicated. Moreover, the fact that our system is embedded in higher-order logic means that we can readily introduce new syntax for common assertions that arise in practice.

6 Simpson's four slot algorithm

In this section we show how to apply the proof system to verify Simpson's four slot algorithm. We looked at several other examples (Peterson's mutual exclusion algorithm and the Linux spin-lock implementation), but Simpson's algorithm is considerably more interesting, and exercises several novel features of our proof system, such as processor assertions that refer to the private state of other processors (essential for the direct proof we give here). Our approach mirrors informal algorithm development for weak memory models: First, we consider the SC case for high-level pseudo-code. Then we incorporate the x86-TSO memory model, and modify the algorithm by including memory barriers. A

```
$\boxed{\text{STATE}}$
$\mathsf{data}[0..1, 0..1]$
$\mathsf{slot}[0..1]$                  // read-only by the reader,  $\mathsf{slot}[\_] \in \{0,1\}$
$\mathsf{latest} = 0$                  // read-only by the reader,  $\mathsf{latest} \in \{0,1\}$
$\mathsf{reading} = 0$                 // read-only by the writer,  $\mathsf{reading} \in \{0,1\}$
$\mathsf{pair}_W,\ \mathsf{index}_W$   // writer local state,       $\mathsf{pair}_W \in \{0,1\},\ \mathsf{index}_W \in \{0,1\}$
$\mathsf{pair}_R,\ \mathsf{index}_R$   // reader local state,       $\mathsf{pair}_R \in \{0,1\},\ \mathsf{index}_R \in \{0,1\}$
$\boxed{\text{WRITER CODE}}$
$\mathsf{pair}_W = \neg\mathsf{reading}$
$\mathsf{index}_W = \neg\mathsf{slot}[\mathsf{pair}_W]$
...//critical section, write to $\mathsf{data}[\mathsf{pair}_W, \mathsf{index}_W]$
$\mathsf{slot}[\mathsf{pair}_W] = \mathsf{index}_W$
$\mathsf{latest} = \mathsf{pair}_W$
$\boxed{\text{READER CODE}}$
$\mathsf{pair}_R = \mathsf{latest}$
$\mathsf{reading} = \mathsf{pair}_R$
$\mathsf{index}_R = \mathsf{slot}[\mathsf{pair}_R]$
...//critical section, read from $\mathsf{data}[\mathsf{pair}_R, \mathsf{index}_R]$
```

Fig. 3. Simpson's four slot algorithm in pseudo-code

key point is that the proof of correctness in the SC case dictates the positioning of the memory barriers in the weak case. Finally we refine the pseudo-code to low-level assembly code. Another key point is that we retain the high-level assertions in the low-level code, however the reasoning is substantially more complicated due to the non-atomic execution of individual instructions.

Simpson's four slot algorithm is designed to ensure mutual exclusion between a single reader and a single writer of a multi-word datum. Simpson's algorithm also satisfies several other desirable properties, but here we focus solely on mutual exclusion. Simpson's algorithm is non-blocking: the reader can still read even when the writer is delayed in the critical section, and vice-versa. This is achieved essentially by maintaining four copies of the underlying data in an array $\mathsf{data}[0..1, 0..1]$ and ensuring that the reader and writer access different slots when running concurrently.

The code in Fig. 3 describes the entry and exit protocol run by the reader and the writer before and after the data array is accessed (the exit protocol for the reader is trivial). This entry and exit code is invoked whenever the writer wants to write to data, or the reader wants to read from data. Thus, the code above could be executed many times. Between executions, arbitrary other code may be executed, but crucially it should not access data.

Simpson's algorithm is correct in the sense that, if the reader and writer are both in their critical sections, then they access different entries in the data array. More formally, we introduce the notation $\mathsf{pc}_R \in \boxdot$ ($\mathsf{pc}_R \in \cdot\Box$) to mean that the reader is (is not) in the critical section. We then have the following:

```
$\boxed{\text{Writer code}}$
  $\mathsf{pair}_W = \neg\mathsf{reading}$
$\{\mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}])\ \}$
  $\mathsf{index}_W = \neg\mathsf{slot}[\mathsf{pair}_W]$
$\{\mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}]),\ \mathsf{index}_W = \neg\mathsf{slot}[\mathsf{pair}_W]\}$
  $\ldots$ //critical section, write to $\mathsf{data}[\mathsf{pair}_W, \mathsf{index}_W]$
  $\mathsf{slot}[\mathsf{pair}_W] = \mathsf{index}_W$
  $\mathsf{latest} = \mathsf{pair}_W$
$\boxed{\text{Reader code}}$
  $\mathsf{pair}_R = \mathsf{latest}$
  $\mathsf{reading} = \mathsf{pair}_R$
$\{\mathsf{pc}_R \in \cdot\Box,\ \mathsf{pair}_R = \mathsf{reading}\ \}$
  $\mathsf{index}_R = \mathsf{slot}[\mathsf{pair}_R]$
$\{\mathsf{pc}_R \in \boxdot,\ \mathsf{pair}_R = \mathsf{reading}\ \}$
  $\ldots$ //critical section, read from $\mathsf{data}[\mathsf{pair}_R, \mathsf{index}_R]$
```

Fig. 4. Annotated pseudo-code for SC

$$\begin{array}{l}
\{\mathsf{pc}_W \in \boxdot \longrightarrow \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W, \mathsf{index}_W) \neq (\mathsf{pair}_R, \mathsf{index}_R)\}\\
\quad = \qquad //\textit{by propositional reasoning}\\
\{\mathsf{pc}_W \in \boxdot \longrightarrow \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{pair}_R) \longrightarrow (\mathsf{index}_R \neq \mathsf{index}_W)\}\\
\quad = \qquad //\textit{since } \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_R = \mathsf{reading})\\
\{\mathsf{pc}_W \in \boxdot \longrightarrow \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R \neq \mathsf{index}_W)\}\\
\quad = \qquad //\textit{since } \mathsf{pc}_W \in \boxdot \longrightarrow (\mathsf{index}_W = \neg\mathsf{slot}[\mathsf{pair}_W])\\
\{\mathsf{pc}_W \in \boxdot \longrightarrow \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}])\}
\end{array}$$

ie the algorithm is correct provided the *main correctness assertion* $\mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}])$ holds in the writer's critical section. Of course, we must also ensure that the auxiliary facts we used in the equality proof above are valid, but this is easy to see from the code. Note that this writer assertion refers to the program counter and private register state of the reader. The annotated code is in Fig. 4.

The writer assertions in Fig. 4 are trivially true for the writer if there is no reader. In the concurrent setting, following Rely-Guarantee [Jon81], we must check that these assertions are true regardless of steps taken by the reader: we check that the properties are closed under interference from the reader. First we express the interference from the reader as a relation between states. The notation $S \rightsquigarrow S'$ represents the relation $S \times S'$. Consider the Fig. 4 annotated code for the reader. Since these statements concern reader thread-local state pc_R and pair_R, or global state $\mathsf{reading}$ that is read-only by the writer, these assertions are closed under writer interference. Examining this annotated code reveals the interference from the reader consists of: Updates to reading outside the critical section: $(\mathsf{pc}_R \in \cdot\Box, \mathsf{reading} = i) \rightsquigarrow (\mathsf{pc}_R \in \cdot\Box, \mathsf{reading} = j),\ i \in \{0,1\},\ j \in \{0,1\}$; Entrance to the critical region:

$(\mathsf{pc}_R \in \cdot\Box, \mathsf{pair}_R = \mathsf{reading}) \rightsquigarrow (\mathsf{pc}_R \in \boxdot, \mathsf{pair}_R = \mathsf{reading}, \mathsf{index}_R = \mathsf{slot}[\mathsf{pair}_R])$; Exit from the critical region: $(\mathsf{pc}_R \in \boxdot) \rightsquigarrow (\mathsf{pc}_R \in \cdot\Box)$. The only non-trivial interference involves the reader entering the critical section, but it is immediate that this preserves the main correctness assertion. A key point is that the reader interference is dependent on which region of code the reader is executing ($\mathsf{pc}_R \in \cdot\Box$, $\mathsf{pc}_R \in \boxdot$). This is the main motivation for our liberal notion of state, which can be incorporated into traditional proof systems unrelated to weak memory. In general, we expect more complicated algorithms will involve many more "regions". The key idea here is to index the rely and guarantee relations by the region in which the code is executing.

Simpson's algorithm for x86-TSO A common approach to adapting algorithms to weak memory models is to insert memory synchronization operations eg memory barriers. One option is to insert barriers between every instruction, which is sufficient to regain SC behaviour for x86-TSO. However, synchronization operations are typically very expensive, so for performance reasons it is important to minimize their use. In this section we show how the SC proof dictates where to place memory barriers in the weak case.

We first examine the interference from the reader, specifically interference when entering the critical section. The first reader assertion in Fig. 4 states that the value of the reader's thread-local register pair_R is equal to the main memory value at address $\mathsf{reading}$. Unfortunately, since writes to memory may be buffered, this assertion no longer holds. The fix is to insert a memory barrier after the write to $\mathsf{reading}$.

Now consider the first writer assertion in Fig. 4. Whilst the assertion is closed under interference from the reader, the writer's own write buffer may asynchronously flush a write to memory which invalidates the assertion. For example, there may be a write to $\mathsf{slot}[\mathsf{pair}_W]$ from a previous execution of the writer's exit protocol which is still in the write buffer. The problem is that the assertion is not closed under write buffer transitions, as required by the proof system. The simplest fix is to ensure that there are no pending writes in the write buffer to addresses which are involved in the assertion. The assertion mentions two addresses, $\mathsf{reading}$ and $\mathsf{slot}[\mathsf{pair}_W]$. The write buffer will never contain writes to $\mathsf{reading}$ since it is read-only by the writer. Thus, an invariant for the writer is the assertion $\{\mathsf{reading} \doteq []\}$. To rule out the possibility of a write to slot from a previous execution of the writer exit protocol, we can insert a memory barrier at the end of the writer code, see Fig. 5. The alternative, placing the barrier immediately after the write to $\mathsf{slot}[\mathsf{pair}_W]$ means that the write to latest may be delayed, potentially reducing performance, though not correctness.

Simpson's algorithm in x86 assembly code with x86-TSO We now refine the high-level pseudo-code of the preceding section to low-level x86 assembly code. This makes the whole development more realistic, but the length of the code increases dramatically, and although high-level assertions are preserved at the low-level, there are now many intermediate assertions which obscure the main correctness argument.

```
$\boxed{\textsc{Writer code}}$
$\{\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq []\ \}$
    $\mathsf{pair}_W = \neg\mathsf{reading}$
$\{\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq [],\ \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}])\}$
    $\mathsf{index}_W = \neg\mathsf{slot}[\mathsf{pair}_W]$
$\{\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq [],\ \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}])\}$
    $\ldots//\textit{critical section, write to } \mathsf{data}[\mathsf{pair}_W, \mathsf{index}_W]$
    $\mathsf{slot}[\mathsf{pair}_W] = \mathsf{index}_W$
    $\mathsf{latest} = \mathsf{pair}_W$
    barrier mfence
$\{\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq []\ \}$
```

Fig. 5. Annotated pseudo-code for x86-TSO

The writer pseudo-code starts with $\mathsf{pair}_W = \neg\mathsf{reading}$, which translates to the three x86 assembly instructions in Fig. 6. The initial and final assertions are exactly those of the pseudo-code version, and the expected intermediate assertions do indeed hold, but for far from obvious reasons. Interested readers may consult the formal development for further details. To clarify the exposition, we suppress the trivial assertions $\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq []$ in these intermediate assertions. The remaining pseudo-code instructions are tackled in a similar way.

7 Related work

We have included references to the main sources in the body of the text. The x86-TSO memory model [OSS09] was introduced in two provably-equivalent styles, axiomatic and operational, but here it suffices to consider only the operational model (called the "abstract machine memory model" in [OSS09]). In order to make our development self-contained, we have reproduced this memory model, and in addition incorporated a model of x86 instructions. It would be reasonably straightforward to establish a formal connection between our model and x86-TSO. Our x86 instruction model is based on that of Myreen [MSG08] which has been extensively validated in the sequential case; we believe the model is accurate for the concurrent case considered here. Whilst the model is not intended to be exhaustive, it should be straightforward to extend it. One of the challenges of this work was dealing with non-atomic x86 assembly instructions. Similar issues are the subject of ongoing research [Col08].

Our proof for Simpson's algorithm in the SC case is new, as far as we are aware. Many other proofs for SC exist in the literature eg [Hen03,Rus02]. As discussed in previous sections, our proof is unsuited to traditional program logics because of the need to refer to the private state of other processors. There are several other approaches to ensuring correctness of programs running on weak memory models. Data-race free (DRF) programs can be reasoned about using

$\boxed{\text{Writer code}}$

$\{\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq []\ \}$

$\quad \mathtt{mov}_{\mathsf{rm}}(\mathsf{eax},\ \mathsf{reading})$

$\{\mathsf{pc}_R \in \boxdot \longrightarrow (1\ -\ \mathsf{eax} = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}\ (\mathsf{reading}))\ \}$

$\quad \mathtt{mov}_{\mathsf{ri}}(\mathsf{pair}_{\mathsf{W}},\ 1)$

$\{\mathsf{pc}_R \in \boxdot \longrightarrow (1\ -\ \mathsf{eax} = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}\ (\mathsf{reading})),\ \mathsf{pair}_{\mathsf{W}} = 1\ \}$

$\quad \mathtt{sub}_{\mathsf{rr}}(\mathsf{pair}_{\mathsf{W}},\ \mathsf{eax})$

$\{\mathsf{reading} \doteq [],\ \mathsf{slot}[\_] \doteq [],\ \mathsf{pc}_R \in \boxdot \longrightarrow (\mathsf{pair}_W = \mathsf{reading}) \longrightarrow (\mathsf{index}_R = \mathsf{slot}[\mathsf{reading}])\}$

Fig. 6. Annotated x86 assembly code for x86-TSO

SC techniques. A strengthening of DRF techniques to x86-TSO is [Owe10]. An interesting approach that shares some similarities with DRF techniques is [CS09]. Model checking is another technique that has been fruitfully applied in this area [PD95], and one can always resort to direct operational proofs [Rid07].

8 Conclusion

We presented a proof system for concurrent low-level assembly code and the x86-TSO memory model. Some features of the proof system, such as the liberal notion of state, are independent of the memory model and may find use elsewhere. Some features are specific to x86-TSO, such as the need to use assertions closed under write buffer interference (and the practical importance of having a syntactic "closure under write buffer interference" operation) and how to incorporate a smooth treatment of write buffers as degenerate processes.

Mechanization revealed several unexpected areas for future work. For example, intermediate assertions at the assembly code level involved substantially more complicated proofs than at higher-levels, although intuitively the proof effort should be similar. One possible explanation is that our proof system can distinguish intermediate states in the execution of a single instruction, however in practice assertions do not make such distinctions. Therefore one may expect that the proof system can be simplified by making further assumptions about the nature of assertions. Clearly there is a trade-off here between the complexity of the judgement semantics and the complexity of the proof rules: in this paper we have chosen to keep the judgement semantics as simple as possible, but other choices are certainly possible. Our argument for the correctness of the high-level pseudo-code running against x86-TSO should be made formal, by taking an appropriate model of a high-level language (eg Xavier Leroy's Clight [BL09]) and exposing the low-level memory model. This would involve tracking the memory model through the different stages of the compilation process. A much easier alternative would be to design an operational semantics of a high-level language that incorporates x86-TSO from scratch, and carry out the proofs of correctness against this model.

We believe that our proof system makes the presentation of proofs much more palatable. However, the reasoning is still very low-level and operational, and creating a proof takes significant effort. Having talked with low-level programmers, it appears that most think very operationally about these programs, and that few high-level abstractions or concepts have emerged. One can incorporate other orthogonal abstractions such as separation logic but it is not clear that this would make these programs essentially easier to reason about. This work has uncovered several key requirements, but a key challenge remains: to establish higher-level notions for reasoning about programs executing with relaxed memory models. The author acknowledges funding from EPSRC grants EP/F036345 and EP/F019394.

References

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *CoRR*, abs/0901.3619, 2009.

[CH72] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1:214–224, 1972.

[Col08] Joey W. Coleman. Expression decomposition in a Rely/Guarantee context. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, pages 146–160, 2008.

[CS09] Ernie Cohen and Norbert Schirmer. A better reduction theorem for store buffers. Technical report, 2009.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. American Mathematical Society Symposia in Applied Mathematics 19*, 1967.

[Hen03] Neil Henderson. Proving the correctness of Simpson's 4-slot ACM using an assertional Rely-Guarantee proof method. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2003.

[Hoa69] Hoare. An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 12, 1969.

[Jon81] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Prgr.Res.Grp. 25, Oxford Univ., Comp. Lab., UK, June 1981.

[Lin99] 1999. Linux Kernel mailing list, thread "spin_unlock optimization(i386)", 119 messages, Nov. 20–Dec. 7th, http://www.gossamer-threads.com/lists/engine?post=105365;list=linux. Accessed 2009/11/18.

[MSG08] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures: An application of decompilation into logic. In *Proc. FMCAD*, 2008.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs, LNCS 5674*, pages 391–407, 2009.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, 2010.

[PD95] Park and Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[Rid07] Tom Ridge. Operational reasoning for concurrent Caml programs and weak memory models. In *Proc. TPHOLs, LNCS 4732*, pages 278–293, 2007.

[Rus02] John Rushby. Model checking Simpson's four-slot fully asynchronous communication mechanism, 2002.

[Sim90] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Computers and Digital Techniques*, 137(1):17–30, January 1990.