Software Verification for Weak Memory via Program Transformation*

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig

University College London and University of Oxford

Abstract Multiprocessors implement weak memory models, but program verifiers often assume *Sequential Consistency* (SC), and thus may miss bugs due to weak memory. We propose a sound transformation of the program to verify, enabling SC tools to perform verification w.r.t. weak memory. We present experiments for a broad variety of models (from x86-TSO to Power) and a vast range of verification tools, quantify the additional cost of the transformation and highlight the cases when we can drastically reduce it. Our benchmarks include work-queue management code from PostgreSQL.

1 Introduction

Current multi-core architectures such as Intel's x86, IBM's Power or ARM implement *weak memory models* for performance reasons, allowing optimisations such as *instruction reordering*, *store buffering* or *write atomicity relaxation* [3]. These models make concurrent programming and debugging extremely challenging, because the execution of a concurrent program might not be an interleaving of its instructions, as would be the case on a Sequentially Consistent (SC) architecture [22]. As an instance, the lock-free signalling code in the open-source database PostgreSQL failed on regression tests on a PowerPC cluster, due to the memory model. We study this bug in detail in Sec. 5.

This observation highlights the crucial need for weak memory aware verification. Yet, most existing work assume SC, hence might miss bugs specific to weak memory. Recent work addresses the design or the adaptation of existing methods and tools to weak memory [26,30,18,14,24,12,2], but often focuses on one specific model or cannot handle the write atomicity relaxation of Power/ARM: generality remains a challenge.

Since we want to avoid writing one tool per architecture of interest, we propose a unified method. Given a program analyser handling SC concurrency for C programs, we *transform its input* to simulate the possible non-SC behaviours of the program whilst executing the program on SC. Essentially, we augment our programs with arrays to simulate (on SC) the buffering and caching scenarios due to weak memory.

The verification problem for weak memory models is known to be hard (e.g. non-primitive recursive for TSO), if not undecidable (e.g. for RMO-like models) [10]. This means that we cannot design a *complete* verification method. Yet, we can achieve *soundness*, by implementing our tools in tandem with the design of a proof, and by stressing our tools with test cases reflecting subtle points of the proof.

* Supported by EPSRC project EP/G026254/1 and the Semiconductor Research Coropration (SRC) under task 2269.002.

We also aim for an effective and unified verification setup, where one can easily plug a tool of choice. This paper meets these objectives by making three new contributions:

1. To design our transformation, we define in Sec. 3 a generic abstract machine that we prove (in the Coq proof assistant) equivalent to the framework of [8] (recalled in Sec. 2). We also explain how this equivalence proof allows us to design a drastically improved transformation with a speed-up of more than two orders of magnitude.
2. Sec. 4 describes our implementation, highlighting the generality of our approach: we support a broad variety of models (x86/TSO, PSO, RMO and Power) and program analysers (Blender [21], CheckFence [14], ESBMC [15], MMChecker [18], Poirot [1], SatAbs [16], and Threader [17]).
3. Sec. 5 details our experiments. i) We systematically validate our implementation w.r.t. our theoretical study with 555 *litmus tests* exercising weak memory artefacts. ii) We verify an excerpt of the relational database software PostgreSQL, which has a bug specific to Power. iii) Our transformation easily scales to systems code from the Linux kernel or the Apache HTTP server, and also industrial code.

We provide the source and documentation of our tools, our benchmarks, experimental reports, Coq proofs and their typeset sketches online: www.cs.ox.ac.uk/people/vincent.nimal/instrument/

Related Work We focus here on the *verification* problem, i.e., detecting the behaviours that are buggy, not all the non-SC ones. This problem is non-primitive recursive for TSO [10]. It is undecidable if read/write or read/read pairs can be reordered, as in RMO-like models [10]. Forbidding *causal loops* restores decidability; relaxing write atomicity makes the problem undecidable again [11].

Some solutions use various bounds over the objects of the model [12,20], over-approximate the possible behaviours [21,19], or relinquish termination [23]. For TSO, [2] presents a sound and complete solution. We present a provably sound method that allows to lift any SC method or tool to a large spectrum of weak memory models, ranging from x86 to Power. We build an operational model; [25] presented such a model, but theirs is restricted to TSO. Given the undecidability of the problem, we cannot provide completeness, as we focus on soundness. We do not use any bound in our theoretical model (Sec. 3), but our implementation uses finite buffers (Sec. 4).

Our approach also reduces the amount of instrumentation in a provably sound manner. Unlike [12], we only instrument selected shared memory accesses. For TSO this would follow immediately from [13], but we generalise to models such as Power.

We draw the attention of the reviewers to our TACAS submission [9], which presents a BMC encoding for the verification problem, not based on instrumentation.

2 Context: Axiomatic Memory Model

We use the framework of [8], which provably embraces several *architectures*: SC [22], Sun TSO (i.e. the x86 model [25]), PSO and RMO, Alpha, and a fragment of Power.

We present this framework via *litmus tests*: we give an example in Fig. 1. The left-hand side of the figure shows a multi-threaded program. The shared variables `x` and

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

Final state? `r1=0`; `r2=0`

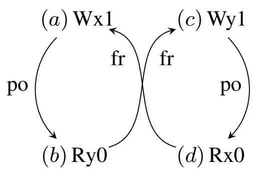


Figure 1. Store Buffering (**sb**)

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x`	(c) `r3` $\leftarrow$ `y`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `1`
(b) `r2` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`		

Final state? `r1=1`; `r2=0`; `r3=1`; `r4=0`;

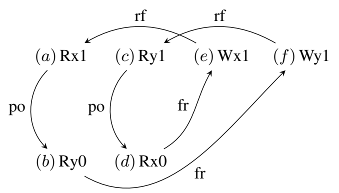


Figure 2. Independent Reads of Independent Writes (**iriw**)

`y` are initialised to zero. The property of interest is whether the program can reach the final state in which `r1=0` and `r2=0`. This can be decided by examining a corresponding *event graph*, given on the right-hand side of the figure. It is composed of relations over *read and write memory events*: a store instruction (e.g. `x` $\leftarrow$ `1` on P_0) gives rise to a write event ((a)Wx1), and a load (e.g. `r1` $\leftarrow$ `y` on P_0) to a read event ((b)Ry0). An architecture allows an execution when it represents a *consensus* amongst the processors. A cycle in an execution graph is a potential violation of this consensus. Thus, if the graph has a cycle, we check if the architecture *may relax* some relations. This might make the graph acyclic, which implies that the architecture allows the final state.

In SC, nothing may be relaxed, thus the cycle in Fig. 1 forbids the execution. On the other hand, x86 may relax the program order (po in Fig. 1) between writes and reads, thus the forbidding cycle no longer exists, and the given final state can be observed.

Executions Formally, an *event* is a read or a write memory access, composed of a unique identifier, a direction R for read or W for write, a memory address, and a value. We represent each instruction by the events it issues. In Fig. 2, we associate the store $x \leftarrow$ `1` on processor P_2 with the event $(e)\text{W}x1$.

We define an *event structure* $E \triangleq (\mathbb{E}, \mathsf{po})$, composed of its events $\mathbb{E}$ and the *program order* po, a per-processor total order. We write dp for the relation (included in po, the source being a read) that models the *dependencies* between instructions, e.g. an *address dependency* occurs when computing the address of a load or store from the value of a preceding load.

We represent the *communication* between processors leading to the final state via an *execution witness* $X \triangleq (\mathsf{ws}, \mathsf{rf})$, which consists of two relations over the events. First, the *write serialisation* ws is a per-address total order on writes which models the *memory coherence* widely assumed by modern architectures. It links a write w to any write w' to the same address that hits the memory after w. Second, the *read-from* relation rf links a write w to a read r such that r reads the value written by w.

We include the writes in the consensus via the write serialisation. Unfortunately, the read-from map does not give us enough information to embed the reads as well. To that aim, we derive the *from-read* relation fr from ws and rf. A read r is in fr with a write w when the write w' from which r reads hit the memory before w did. Formally, we have: $(r, w) \in \mathsf{fr} \triangleq \exists w', (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}$.

In Fig. 2, the specified outcome corresponds to the execution on the right if each memory location initially holds 0. If `r1=1` in the end, the read (a) obtained its value from the write (e) on P_2, hence $(e, a) \in \mathsf{rf}$. If `r2=0` in the end, the read (b) obtained its value from the initial state, thus before the write (f) on P_3, hence $(b, f) \in \mathsf{fr}$. Similarly, we have $(f, c) \in \mathsf{rf}$ from `r3=1`, and $(d, e) \in \mathsf{fr}$ from `r4=0`.

Relaxed or safe A processor can commit a write w first to a store buffer, then to a cache, and finally to memory. When a write hits the memory, all the processors agree on its value. But while the write w is in transit through store buffers and caches, a read r can occur before the value is actually available to all processors from the memory. In this case, the read-from relation between the write w and the read r does not contribute to the consensus, since the reading occurs in advance.

We model this by some subrelation of the read-from rf being *relaxed*, i.e. not included in the consensus. When a processor can read from its own store buffer [3] (the typical TSO/x86 scenario), we relax the internal read-from rfi. When two processors P_0 and P_1 can communicate privately via a cache (a case of *write atomicity* relaxation [3]), we relax the external read-from rfe, and call the corresponding write *non-atomic*. This is the main particularity of Power or ARM, and cannot happen on TSO/x86. Some program-order pairs may be relaxed (e.g. write-read pairs on x86, and all but dp ones on Power), i.e. only a subset of po is guaranteed to occur in this order.

When a relation may not be relaxed, we call it *safe*. Architectures provide special *fence* (or *barrier*) instructions to prevent weak behaviours. Following [8], the relation $\mathsf{fence} \subseteq \mathsf{po}$ induced by a fence is *non-cumulative* when it orders certain pairs of events surrounding the fence, i.e. fence is safe. The relation fence is *cumulative* when it makes writes atomic, e.g. by flushing caches. The relation fence is *A-cumulative* (resp. *B-cumulative*) if $\mathsf{rfe}; \mathsf{fence}$ (resp. $\mathsf{fence}; \mathsf{rfe}$) is safe. When stores are atomic (i.e. rfe is safe), e.g. on TSO, we do not need cumulativity.

Architectures An *architecture* A determines the set safe_A of the relations safe on A, i.e. the relations embedded in the consensus. Following [8], we always consider the write serialisation ws and the from-read relation fr safe. SC relaxes nothing, i.e. rf and po are safe. TSO authorises the reordering of write-read pairs and store buffering (i.e. $\mathsf{po}_{\mathsf{WR}}$ and rfi may be relaxed) but nothing else.

Finally, an execution (E, X) is *valid* on A when the three following conditions hold. 1. SC holds per address, i.e. the communication and the program order for accesses with same address $\mathsf{po\text{-}loc}$ are compatible: $\mathrm{uniproc}(E, X) \triangleq \mathrm{acyclic}(\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{po\text{-}loc})$. 2. Values do not come out of thin air, i.e. there is no causal loop: $\mathrm{thin}(E, X) \triangleq \mathrm{acyclic}(\mathsf{rf} \cup \mathsf{dp})$. 3. There is a consensus, i.e. the safe relations do not form a cycle: $\mathrm{consensus}(E, X) \triangleq \mathrm{acyclic}((\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{po}) \cap \mathrm{safe}_A)$. Formally:

$$\mathrm{valid}_A(E, X) \triangleq \mathrm{uniproc}(E, X) \wedge \mathrm{thin}(E, X) \wedge \mathrm{consensus}(E, X)$$

3 Simulating Weak Behaviours on SC

This section is our first contribution. We are given a weak architecture A, which defines an axiomatic model as described in Sec. 2. We define here an *abstract machine* (Sec. 3.1), featuring a store buffer per address and a load queue. We then show in Sec. 3.3 the equivalence of the axiomatic model of Sec. 2 and the abstract machine. We explain in Sec. 3.4 how this equivalence proof guides our instrumentation strategy.

3.1 Abstract machine

We define a non-deterministic state machine that reads a sequence of *labels*. The machine has a designated bad state $\perp$, and all other states of the machine represent system configurations, i.e. the memory, buffers, and a queue. We write addr, evt, and rln for the types of memory addresses, events and relations, respectively.

Definition 1 (State). *A* state *of the machine is either $\perp$ or a triple $(\mathsf{m}, \mathsf{b}, \mathsf{q})$, where*

- *the* memory m : *addr $\rightarrow$ evt maps a memory address ℓ to a write to ℓ;*
- *the* buffer b : *rln evt is a total order over writes to the same address; the buffer has a special symbol $\perp_{\mathsf{b}}$, placed before all events in the buffer;*
- *the* queue q : *rln (evt $\times$ evt) is a relation over reads, tracking instruction dependencies; the queue has a special symbol $\perp_{\mathsf{q}}$, placed before all events in the queue.*

We have a sole queue, but one buffer per address. Buffers are totally ordered, whereas the queue provides a rather loose ordering. Existing formalisations [25,12] use per-thread buffers, whereas our buffers are solely per-address objects. This allows us to model not only store buffering (which per-thread objects would allow), but also caching scenarios as exhibited by **iriw+dps** (i.e. the **iriw** test of Fig. 2 with dependencies between the reads on P_0 and P_1 to prevent their reordering), i.e. fully non-atomic stores.

The machine performs transitions depending on *delay* and *flush* labels. Intuitively, a delay label pushes an object in the buffer or queue. A flush label makes it exit the buffer or queue. The details of transitions are described below.

Definition 2 (Label). *For a write event w, $\mathrm{d}(\mathrm{w}(w))$ denotes its* delay label*, and $\mathrm{f}(\mathrm{w}(w))$ its* flush label*. For a read event r, its delay label (with direction r, read) is denoted by $\mathrm{d}(\mathrm{r}(w, r))$, and its flush is denoted by $\mathrm{f}(\mathrm{r}(w, r))$.*

A set L of labels is well-formed w.r.t. an event structure E when: in $\mathrm{d}(\mathrm{w}(w))$ or $\mathrm{f}(\mathrm{w}(w))$, w is a write of E; in $\mathrm{d}(\mathrm{r}(w, r))$ or $\mathrm{f}(\mathrm{r}(w, r))$, w is a write of E, r a read of E, both with the same address; any event of E has a unique corresponding flush label in L; when a flush label belongs to L, so does its delay counterpart.

Transitions We write $s \xrightarrow{l} s'$ to denote that the machine can make a transition from state s to state s' reading label l. Let the machine be in a state $(\mathsf{m}, \mathsf{b}, \mathsf{q})$. Given a label, the machine performs transitions from one state to another if the conditions described below are fulfilled. Otherwise, the machine transitions to $\perp$ (it gets stuck).

$$
\begin{aligned}
\mathrm{updm}(\mathsf{m}, w) &\triangleq x \mapsto \text{if } \mathrm{addr}(x) = \mathrm{addr}(w) \text{ then } w \text{ else } x \\
\mathrm{updb}(\mathsf{b}, w) &\triangleq \mathsf{b} \cup \{(w_1, w_2) \mid w_1 = \bot_{\mathsf{b}} \vee ((\bot_{\mathsf{b}}, w_1) \in \mathsf{b} \wedge \mathrm{addr}(w_1) = \mathrm{addr}(w)) \wedge \\
&\quad w_2 = w\} \\
\mathrm{updq}(\mathsf{q}, r) &\triangleq \mathsf{q} \cup \{(r_1, r_2) \mid r_1 = \bot_{\mathsf{q}} \wedge r_2 = r\} \\
\mathrm{delb}(\mathsf{b}, w) &\triangleq \{(w_1, w_2) \mid (w_1, w_2) \in \mathsf{b} \wedge w_1 \neq w\} \\
\mathrm{delq}(\mathsf{q}, r) &\triangleq \{(r_1, r_2) \mid (r_1, r_2) \in \mathsf{q} \wedge r_1 \neq r \wedge r_2 \neq r\} \\
\mathrm{last}(\mathsf{b}, w) &\triangleq \neg(\exists w', (\bot_{\mathsf{b}}, w') \in \mathsf{b}) \wedge w = \bot_{\mathsf{b}}) \vee \\
&\quad ((\exists w', (\bot_{\mathsf{b}}, w') \in \mathsf{b}) \wedge (\bot_{\mathsf{b}}, w) \in \mathsf{b} \wedge \neg(\exists w', (w', w) \in \mathsf{b}))
\end{aligned}
$$

WRITE TO BUFFER

$$\frac{\top}{s \xrightarrow{\mathrm{d}(\mathrm{w}(w))} (\mathsf{m}, \mathrm{updb}(\mathsf{b}, w), \mathsf{q})}$$

ENQUEUE READ

$$\frac{\top}{s \xrightarrow{\mathrm{d}(\mathrm{r}(w,r))} (\mathsf{m}, \mathsf{b}, \mathrm{updq}(\mathsf{q}, r))}$$

WRITE FROM BUFFER TO MEMORY

$$\frac{\begin{array}{c} \mathrm{rr}(\mathsf{b} \cup \mathsf{q}, \{e \mid (e, w) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset \wedge \\ \mathrm{rr}(\mathsf{q}, \{r \mid (r, w) \in \mathsf{po\text{-}loc}\}) = \emptyset \wedge \\ \mathrm{last}(\mathrm{rr}(\mathsf{b}, \{e \mid \mathrm{addr}(e) = \ell\}), w) \end{array}}{s \xrightarrow{\mathrm{f}(\mathrm{w}(w))} (\mathrm{updm}(\mathsf{m}, w), \mathrm{delb}(\mathsf{b}, w), \mathsf{q})}$$

READ FROM QUEUE

$$\frac{\begin{array}{c} r \in \mathsf{q} \wedge \\ \mathrm{rr}(\mathsf{q}, \{r \mid (r, w) \in \mathsf{dp}\}) = \emptyset \wedge \\ \mathrm{rr}(\mathsf{b} \cup \mathsf{q}, \{e \mid (e, r) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset \wedge \\ \big[(w = \mathsf{m}(\mathrm{addr}(r)) \wedge \\ \mathrm{rr}(\mathsf{b}, \{w \mid (w, r) \in \mathsf{po\text{-}loc}\}) = \emptyset\big) \vee \\ (w \in \mathsf{b} \wedge \mathrm{visible}(w, r))\big] \end{array}}{s \xrightarrow{\mathrm{f}(\mathrm{r}(w,r))} (\mathsf{m}, \mathsf{b}, \mathrm{delq}(\mathsf{q}, r))}$$

Figure 3. The abstract machine

In Fig. 3, we give the formal definition of the transitions of our machine. We need to define a few auxiliary functions, also formally defined in Fig. 3. We update the memory with a write w via $\mathrm{updm}(\mathsf{m}, w)$, a buffer with a write w via $\mathrm{updb}(\mathsf{b}, w)$, and a queue with a read r via $\mathrm{updq}(\mathsf{q}, r)$. We delete a write w from a buffer via $\mathrm{delb}(\mathsf{b}, w)$ and we delete a read r from a queue via $\mathrm{delq}(\mathsf{q}, r)$. We write $\mathrm{rr}(R, S)$ for the restriction of a relation R to a set S, i.e. $\{(x, y) \mid (x, y) \in R \wedge x \in S \wedge y \in S\}$. We pick the last write to an address ℓ of a buffer via $\mathrm{last}(\mathsf{b}, w)$. In prose, the transitions are as follows:

- *Write to buffer*: a write $\mathrm{d}(\mathrm{w}(w))$ to address ℓ can always enter the buffer b, taking its place after all the writes to ℓ that are already in b ($\mathrm{updb}(\mathsf{b}, w)$).
- *Write from buffer to memory*: a write $\mathrm{f}(\mathrm{w}(w))$ to address ℓ exits the buffer b ($\mathrm{delb}(\mathsf{b}, w)$) and updates the memory at ℓ ($\mathrm{updm}(\mathsf{m}, w)$) if:
  - there is no event e in $\mathsf{ppo} \cup \mathsf{ab}$ before w – neither in the buffer nor in the queue ($\mathrm{rr}(\mathsf{b} \cup \mathsf{q}, \{e \mid (e, w) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset$);
  - *and* there is no read from ℓ in po before w in the buffer ($\mathrm{rr}(\mathsf{q}, \{r \mid (r, w) \in \mathsf{po\text{-}loc}\}) = \emptyset$);
  - *and* there is no write to ℓ before w in b ($\mathrm{last}(\mathrm{rr}(\mathsf{b}, \{e \mid \mathrm{addr}(e) = \ell\}), w)$).

– *Enqueue read*: a read $\mathrm{d}(\mathrm{r}(w, r))$ can always enter the queue q $(\mathrm{updq}(\mathsf{q}, r))$.
– *Read from queue*: a read $\mathrm{f}(\mathrm{r}(w, r))$ from ℓ exits the queue $(\mathrm{delq}(\mathsf{q}, r))$ if:
  - there is no read in dp before w in the queue $(\mathrm{rr}(\mathsf{q}, \{r \mid (r, w) \in \mathsf{dp}\}) = \emptyset)$;
  - *and* there is no event in $\mathsf{ppo} \cup \mathsf{ab}$ before r in the buffer nor in the queue $(\mathrm{rr}(\mathsf{b} \cup \mathsf{q}, \{e \mid (e, w) \in \mathsf{ppo} \cup \mathsf{ab}\})) = \emptyset)$;
  - *and either* w is in memory $(w = \mathsf{m}(\mathrm{addr}(r)))$, and there is no write to ℓ before r in po in the buffer $(\mathrm{rr}(\mathsf{b}, \{w \mid (w, r) \in \textsf{po-loc}\}) = \emptyset)$;
  - *or* w is in the buffer and is *visible to* r (a notion defined below).

To define a write w as *visible to a read* r, we need a few auxiliary functions. We define the part of the buffer visible to a read r as follows: $\mathsf{b}_r \triangleq \{w \mid (\perp_{\mathsf{b}}, w) \in \mathsf{b} \wedge((\mathsf{rfi} \subseteq \mathrm{safe}_A) \Rightarrow \mathrm{proc}(w) = \mathrm{proc}(r))\wedge((\mathsf{rfe} \subseteq \mathrm{safe}_A) \Rightarrow \mathrm{proc}(w) \neq \mathrm{proc}(r))))\}$. Now, w is visible to r when:

w and r share the same address ℓ;
w is in the part of the buffer visible to r a thread whose buffer r can read w.r.t. A, namely if rfi (resp. rfe) is safe then w cannot be on the same thread as A (resp. a different) thread as r $(w \in \mathsf{b}_r)$;
w is in the buffer before the first write to w_a ℓ in po after r $((\perp_{\mathsf{b}}, w_a) \in \mathsf{b} \wedge (r, w_a) \in \textsf{po-loc} \wedge \neg(\exists w', (r, w') \in \textsf{po-loc} \wedge (w', w) \in \textsf{po-loc}))$ and $(w, w_a) \in \mathsf{b})$;
w is equal to, or in the buffer after, the last write w_b to ℓ in po before r $((\perp_{\mathsf{b}}, w_b) \in \mathsf{b} \wedge (w_b, r) \in \textsf{po-loc} \wedge \neg(\exists w', (w_b, w') \in \textsf{po-loc} \wedge (w', r) \in \textsf{po-loc}))$.

All states except $\perp$ are accepting states. Thus, the abstract machine accepts a sequence p of labels $l_0, l_1, \ldots$ if there is a sequence of states $s_0, s_1, \ldots$ such that $s_i \xrightarrow{l_i} s_{i+1}$ and $s_i \neq \perp$ for all i.

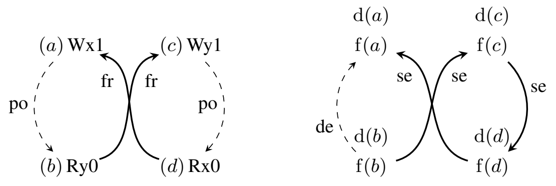
Definition 3 (Accepting sequence). *A* sequence *p is a total order over L compatible with the program order, i.e. for two events $(x, y) \in \mathsf{po}$, their delay labels appear in the same order in p. It is* accepting *iff the sequence p is accepted by the abstract machine.*

3.2 Illustration using Examples

We illustrate the machine by revisiting the **sb** test of Fig. 1 for TSO and the **iriw** test of Fig. 2 for Power.

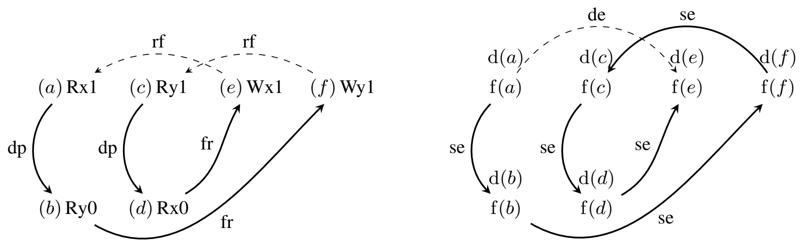
Fig. 4 and 5 reproduce on the left the event graphs from Fig. 1 and 2. On the right, they show the counterparts in the abstract machine. We explain the labels on the arrows in the next section (§"From the axiomatic model to the machine"). We use the following graphical conventions. In the axiomatic world (i.e. on the left of our figures), we reflect a pair that can be relaxed by a dashed arrow. For example, in the **sb** test of Fig. 4 on TSO, the write-read pairs (a, b) and (d, c) can be relaxed. Likewise, in the **iriw+dps** test of Fig. 5 on Power, the read-from pairs (e, a) and (f, c) can be relaxed (as opposed to the read-read pairs (a, b) on P_0 and (c, d) on P_1, which are safe because of dependencies).

In any given execution, the abstract machine may relax any pair that is not safe. Such pairs are depicted with a dashed arrow. Pairs that the machine does not relax are depicted with a thick arrow.



(a) Axiomatic model (b) Machine

Figure 4. Revisiting **sb** on TSO with our machine



(a) Axiomatic model (b) Machine

Figure 5. Revisiting **iriw+dps** on Power with our machine

In Fig. 1, the pairs (a, b) on P_0 and (c, d) on P_1 may be relaxed on TSO. Our machine simulates the TSO behaviour by following e.g. the scenario in Fig. 4(b), which corresponds to the path $\mathrm{d}(a) \rightarrow \mathrm{d}(b) \rightarrow \mathrm{d}(c) \rightarrow \mathrm{d}(d) \rightarrow \mathrm{f}(b) \rightarrow \mathrm{f}(c) \rightarrow \mathrm{f}(d) \rightarrow \mathrm{f}(a)$. The machine buffers or enqueues all events w.r.t. program order. In this scenario, the machine chooses to relax the pairs (a, b) by flushing the read b before the write a, ensuring that the registers `r1` and `r2` hold 0 in the end.

In Fig. 2, assume dependencies between the reads on P_0 and P_1, so that (a, b) on P_0 and (c, d) on P_1 are safe on Power. Yet (e, a) and (f, c) may be relaxed on Power, because Power has non-atomic writes. Our machine simulates the weak behaviour exhibited on Power, by following e.g. Fig. 5(b), which corresponds to the path $\mathrm{d}(e) \rightarrow \mathrm{d}(a) \rightarrow \mathrm{f}(a) \rightarrow \mathrm{d}(b) \rightarrow \mathrm{f}(b) \rightarrow \mathrm{d}(f) \rightarrow \mathrm{f}(f) \rightarrow \mathrm{d}(c) \rightarrow \mathrm{f}(c) \rightarrow \mathrm{d}(d) \rightarrow \mathrm{f}(d) \rightarrow\rightarrow \mathrm{f}(e)$. The machine enqueues or buffers all events w.r.t. program order. Since (a, b) and (c, d) are safe on Power, our machine flushes a before b (resp. c before d). Since $(b, f) \in \mathsf{fr}$ (resp. $(d, e) \in \mathsf{fr}$), which is always safe, the machine flushes b before f (resp. d before e), ensuring that b and d read from memory, thus `r2` and `r4` hold 0 in the end. Finally, in this scenario, the machine chooses to relax the pairs (e, a) by flushing a before e, ensuring that `r1` and `r3` hold the value 1 in the end.

3.3 Equivalence of the axiomatic model and the abstract machine

We now prove the equivalence of the axiomatic model of Sec. 2 and the machine defined in Sec. 3.1. We first show that we can build an execution valid in the axiomatic model from any path of labels accepted by the machine (Thm. 1). We then show that we can build a path of labels accepted by the machine from any execution that is valid in axiomatic model (Thm. 2).

Thm. 1 (From the machine to the axiomatic model). *For E and L well-formed w.r.t. E, if there is an accepting sequence for L, there exists an execution witness valid for E.*

Let $\mathrm{ptoX}(p, L)$ denote the execution witness of Thm. 1. Intuitively, we build it as follows. The write serialisation gathers the pairs of writes to the same address according to the order of their flushed parts in p: $\{(w_1, w_2) \mid \mathrm{addr}(w_1) = \mathrm{addr}(w_2) \wedge (\mathrm{f}(\mathrm{w}(w_1)), \mathrm{f}(\mathrm{w}(w_2))) \in p\}$. For the read-from map, we simply gather the pairs given by the labels of L: $\{(w, r) \mid \mathrm{addr}(w) = \mathrm{addr}(r) \wedge \mathrm{f}(\mathrm{r}(w, r)) \in L\}$.

Proof (Thm. 1). We need to show that $(E, \mathrm{ptoX}(p, L))$ passes the $\mathrm{uniproc}$, thin and $\mathrm{consensus}$ checks. The three proofs follow the same lines, thus we focus on the first for brevity.

The execution passes the $\mathrm{uniproc}$ check iff for all $(x, y) \in \mathsf{po\text{-}loc}$, we do not have $(y, x) \in \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{ws} \cup (\mathsf{ws}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf})$ [4, App. A]. By contradiction take $(x, y) \in \mathsf{po\text{-}loc}$ and $(y, x) \in \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{rf}$. We proceed by case disjunction over $(y, x) \in \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{ws} \cup (\mathsf{ws}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf})$. We write ℓ for the address shared by x and y.

If $(y, x) \in \mathsf{rf}$, $\mathrm{f}(\mathrm{r}(y, x)$ is in L. Since p is accepting, the Read from queue transition on $\mathrm{f}(\mathrm{r}(y, x))$ does not block. Hence y is in memory, or y is in the buffer and visible to x. If y is in memory, y has been flushed, i.e. the Write from buffer to memory transition on $\mathrm{f}(\mathrm{w}(y))$ did not block. Hence there is no read from ℓ in po before y in the queue. Yet $(x, y) \in \mathsf{po\text{-}loc}$, and x is still in the queue when y is in memory, a contradiction. If y is in the buffer and visible to x, y is in the buffer before the first write to ℓ in po after x. Yet, $(x, y) \in \mathsf{po\text{-}loc}$, a contradiction.

For brevity, we present only the rf case; all the other cases are similar, using the premises of the rules of the machine. For example the $(y, x) \in \mathsf{ws}$ case uses the Write from buffer to memory rule, in particular the fact that y exits the buffer if there is no write to ℓ before it in the buffer; yet x is still in there. The $(y, x) \in \mathsf{fr}$ case uses the Read from queue rule, in particular the fact that if the write w from which x reads is in memory, then there is no write to ℓ before y in po in the buffer; yet x is in there. If w is in the buffer, we use the fact that w is equal to, or in the buffer after, the last write to ℓ in po before x, which will block the flush of w, a contradiction. $\square$

For the other direction, we first build labels from the events of E. We augment our events with directions: a write w becomes $\mathrm{w}(w)$ and r becomes $\mathrm{r}(w, r)$, where $(w, r) \in \mathsf{rf}$. Then we *split* an augmented event e into its delayed part $\mathrm{d}(e)$, and its flushed part $\mathrm{f}(e)$. We write $\mathrm{labels}(E, X)$ for the labels built from the events of E.

Then we form the *delay pairs* of (E, X), as follows. We build the relation ndelay over the events of E, such that: $((\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr}) \cap \mathrm{safe}_A) \subseteq \mathsf{ndelay}$; ndelay is transitive; ndelay is irreflexive; if $(x, y) \notin \mathsf{ndelay}$ then $(y, x) \in \mathsf{ndelay}$. The delay pairs are simply the pairs (x, y) of events that are not in ndelay.

Given (E, X) and a choice of delay pairs, we build an accepting path p as follows, with e, e_1, and e_2 augmented events:

Delay before flush we always delay an event e before we flush it, i.e. $(\mathrm{d}(e), \mathrm{f}(e)) \in p$;
Enter $(e_1, e_2) \in \mathsf{po}$ enter the buffer or queue in this order, i.e. $(\mathrm{d}(e_1), \mathrm{d}(e_2)) \in p$;
Rf a write enters before we flush a read from it, i.e. $(\mathrm{d}(e_1), \mathrm{f}(e_2)) \in p$ if $(e_1, e_2) \in \mathsf{rf}$;
Safe Exit $(e_1, e_2) \in \mathsf{ndelay}$ are flushed in the same order, i.e. $(\mathrm{f}(e_1), \mathrm{f}(e_2)) \in p$.
Delay Exit $(e_1, e_2) \notin \mathsf{ndelay}$ are flushed in the opposite order, i.e. $(\mathrm{f}(e_2), \mathrm{f}(e_1)) \in p$.

Recall the labels on the arrows of Fig. 4(b) and 5(b): the label "se" corresponds to a safe exit, and "de" to a delay exit. We omit the arrows corresponding to the first three cases to ease the reading of the figures. In Fig. 4(b), we chose (a, b) and (c, d) to be delay pairs, hence we flush them b before a and d before c, following the delay exit rule. On the contrary, (b, c) and (d, a) are not delay pairs, hence we flush b before c and a before d, following the safe exit rule. The same explanation applies in Fig. 5 to the pairs (e, a) and (f, c) being delayed, and (a, b) and (c, d) being safe.

We build $\mathrm{Xtop}(E, X, \mathsf{ndelay})$ as above. Then one can show that since ndelay is acyclic, so is $\mathrm{Xtop}(E, X, \mathsf{ndelay})$. Hence the transitive closure $(\mathrm{Xtop}(E, X, \mathsf{ndelay}))^+$ is a partial order of the labels. Any linearisation $\mathrm{lin}((\mathrm{Xtop}(E, X, \mathsf{ndelay}))^+)$ of this transitive closure forms an actual path, which we show accepting when 1. X is valid 2. this linearisation has finite prefixes, in which case we say that (E, X) has finite prefixes:

Thm. 2 (From the axiomatic model to the machine). *For any valid execution (E, X) with finite prefixes, there is an accepting path p over labels L well-formed w.r.t. E.*

Proof. We need to show that no transition can block the machine. The Write to buffer and Enqueue read transitions are trivial since they can never block.

For the Write from buffer to memory case, suppose as a contradiction that the transition blocks on a write w to an address ℓ. If there is e in $\mathsf{ppo} \cup \mathsf{ab}$ before w in the buffer or the queue, (e, w) cannot be a delay pair (because ppo and ab are safe), i.e. should be flushed in order, contradicting the presence of e in the buffer or the queue. Otherwise, if there is in the queue a read r from ℓ in po before w. Therefore (r, w) is in fr, thus safe, hence cannot be a delay pair, and the same argument applies. Finally, if there is a write w' to ℓ before w in the buffer; one can show that (w', w) is in ws, hence w' should be flushed before w, a contradiction.

For the Read from queue case, suppose as a contradiction that the transition blocks on a read (w, r) with address ℓ. If there is a read r' in dp before w in the queue, one can show that r' should be flushed before r, and r should be flushed before r' (i.e. a thin-air cycle in X), a contradiction. If there is an event in $\mathsf{ppo} \cup \mathsf{ab}$ before r in the buffer or the queue, the reasoning is the same as above in the write case. If w is in memory and there is a write to ℓ before r in po in the buffer, we create a $\mathrm{uniproc}$ cycle, a contradiction. If w is in the buffer and not visible to r, there are two cases. Either w is not on a thread whose buffer r can read w.r.t. A, in which case (w, r) do not form a delay pair and should be flushed in this order, contradicting the presence of w in the buffer. Or w is in the buffer after the first write to ℓ in po after r (or before the last write to ℓ in po before r), in which case we create a uniproc cycle. □

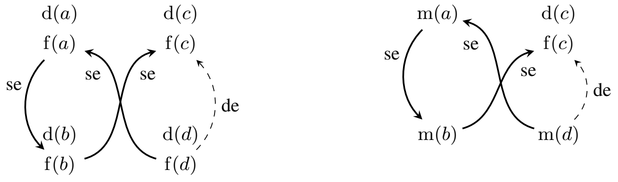
3.4 Instrumentation

Thm. 2 leaves freedom in the instrumentation strategy. We can exploit the choice of delay pairs and the choice of the linearisation of $\mathrm{Xtop}(E, X)$ in order to reduce the overhead of running or verifying an instrumented program.

Choice of delay pairs The conditions on the ndelay relation restrict the choice of delay pairs. We have to put at least all the safe pairs into ndelay, by the first condition.

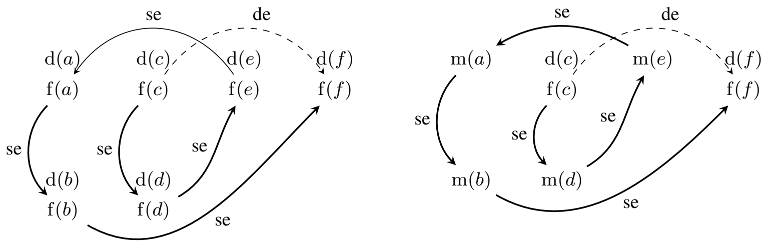
Since ndelay is transitive and irreflexive, it is acyclic. An execution (E, X) presents a cycle iff it is not SC (if it is SC, all pairs are safe and there is no cycle). [7, Thm.1] shows that an execution is valid on A but not on SC iff it contains *critical cycles*[1]. Thus we can put all pairs in ndelay, except one unsafe pair per critical cycle, which corresponds to the last condition over ndelay.

[1] We recall here the definition of [7]. Two events (x, y) are *competing*, written $(x, y) \in \mathsf{cmp}$ if they are from distinct processors, to the same address, and at least one of them is a write (e.g. in



(a) Machine ((b, c) delay) (b) One pair only

Figure 6. Choices for instrumenting **sb** for TSO



(a) (f, c) delay (b) One pair only

Figure 7. Choices for instrumenting **iriw+dps** for Power

In Fig. 4(b), we build an accepting path corresponding to the axiomatic execution of Fig. 4(a) by choosing the unsafe pair (a, b) on the cycle to be a delay. In Fig. 6(a), we choose the unsafe pair (c, d). Similarly for Fig. 5(a), we can build an accepting path corresponding to the axiomatic execution of Fig. 5(a) by choosing e.g. (e, a) as delay (cf. Fig. 5(b)). In Fig. 7(a), we choose (f, c) as delay.

Our examples are symmetric, thus the choice of which pair to delay should not make a difference. In Fig. 1, (a, b) and (c, d) are write-read pairs. Similarly in Fig. 2, (e, a) and (f, c) are of the same nature, namely rfe pairs. For asymmetric examples, the chosen delayed pair can make a crucial difference (cf. Sec. 5), if the instrumentation of one pair causes more execution or verification time overhead than the other.

Choice of the linearisation Thm. 2 accepts any linearisation of $(\mathrm{Xtop}(E, X, \mathsf{ndelay}))^+$. Yet, some require less instrumentation than others. Consider Fig. 6(a) and (b): in both we choose to delay the pair (c, d). On the left, we can pick any interleaving (compatible with Xtop) of the delayed and flushed events to instantiate Thm. 2, e.g. $\mathrm{d}(a) \to \mathrm{d}(b) \to \mathrm{d}(c) \to \mathrm{d}(d) \to \mathrm{f}(b) \to \mathrm{f}(d) \to \mathrm{f}(c) \to \mathrm{f}(a)$.

Fig. 2, the read (a) from x on P_0 and the write (e) to x on P_2). A cycle $\sigma \subseteq \mathsf{cmp} \cup \mathsf{po}$ is critical when it is not a cycle in $(\mathsf{cmp} \cup \mathsf{ppo}_A)^+$ and it satisfies the two following properties: **(i)** Per processor, there are at most two memory accesses (x, y) on this processor and $\mathrm{addr}(x) \neq \mathrm{addr}(y)$. **(ii)** For a given memory address x, there are at most three accesses relative to x, and these accesses are from distinct processors ($(w, w') \in \mathsf{cmp}$, $(w, r) \in \mathsf{cmp}$, $(r, w) \in \mathsf{cmp}$ or $\{(r, w), (w, r')\} \subseteq \mathsf{cmp}$). In Fig. 2, the execution of **iriw** has a critical cycle on Power.

On the right, we write $\mathrm{m}(e)$ when the delayed and flushed part of an event happen without intervening events in between. Observe that in this case, the event e occurs w.r.t. memory: if it is a read, it reads from the memory; if it is a write, it writes to memory. In Fig. 6(b), we pick a particular interleaving, namely the one where all events are w.r.t. memory, except for the event c. This interleaving requires to instrument only one instruction, as opposed to all of them on the left.

Similarly in Fig. 7(a) and (b), we choose in both cases to delay the pair (f, c). On the left, we instrument all instructions. On the right, we instrument only the pair (f, c).

4 Implementation

4.1 Overview

We implemented the transformation technique of Sec. 3. Our tool reads a concurrent C program, possibly with inline assembly `mfence`, `sync`, or `lwsync` instructions (cf. Sec. 2). It generates a new concurrent C program augmented with C equivalents of buffers and queues of Sec. 3.1. The transformation proceeds in three main steps:

1. We devise an *abstract event structure*, as defined below, the concretisation of which amounts to all event structures (cf. Sec. 2) of the program.
2. Given an architecture, we identify potential critical cycles in this structure.
3. We instrument unsafe pairs in the cycle, as described in Sec. 3.4.

The resulting program is then passed to any SC program analyser.

The first two steps guide the program transformation of the third step, in order to reduce the overhead for subsequent verification. As our experiments confirm (Sec. 5), we drastically improve verification performance over instrumenting all instructions.

4.2 Abstract event structures

As described in Sec. 3, we can choose to delay only one pair per critical cycle. To do so, all critical cycles need to be identified first. Sec. 2 defines cycles over events and event structures, which use concrete addresses and values, and thus correspond to concrete execution traces. As the enumeration of all traces is infeasible, we compute a conservative, over-approximate set of possible cycles using static analysis. In this program analysis we introduce *abstract events*, which summarise all concrete events that have the same process identifier, program counter, direction and memory address. We extend the definition of event structure to *abstract event structures*, which are identical except that they use abstract events.

Statements to abstract events The derivation of an abstract event structure from a non-branching multi-threaded program is straight-forward. For each thread, decompose each statement into abstract events, extracting all writes or reads of shared memory. For an assignment to a location designated by a pointer variable, consider the example `*(&x+z) = y;`, where `&x` denotes the address of `x` and `*p` the value held at address `p`. We first read `y`, then read `z` and finally we write to the object pointed to by

&x+z, which is determined using an alias analysis[2]. If the precision of the alias analysis is insufficient to determine the object, we assume that this write can target any of the objects in the program.

```
void* thread_1(void*) {
  int r1;
  x = 2;
  r1 = y;
  y = 2;
}
void* thread_2(void*) {
  int r2, r3;
  y = 1;
  r2 = z;
  r3 = x;
}
```

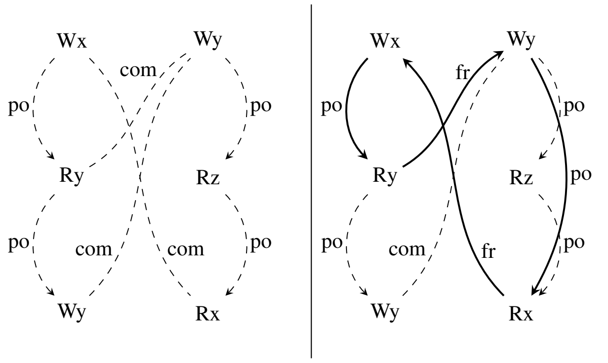


Figure 8. The program on the left contains an **sb** cycle (cf. Fig. 1). We build the abstract event graph in the middle, and indeed detect the cycle in the graph, on the right.

Abstract event graph In order to devise SC cycles that become critical cycles on a weaker architecture, we look for cycles in $\mathsf{ws} \cup \mathsf{fr} \cup \mathsf{rf} \cup \mathsf{po}$ (definition of SC, [5, Thm. 3]). Abstract events in each thread are ordered by program order, po, which we derive as described below. As we do not use concrete values, we compute over-approximations of the relations ws, rf and fr. We further abstract from directed edges and use undirected edges in these over-approximations. We call the abstract event structure equipped with over-approximations of ws, rf and fr an *abstract event graph*. We compute the over-approximations as follows:

- the internal rf, fr and ws pairs (relating two events on the same thread) are already covered by po edges;
- the external rf, fr and ws pairs (relating two events from different threads) are abstracted by undirected external communications, denoted by com, and relate any pair of write-read, read-write or write-write between two distinct threads.

Fig. 8 depicts this first step in the middle, which is the resulting abstract event graph of the program shown on the left-hand side. A concretisation of the abstract event graph may yield critical cycles. Fig. 8 shows an example of a critical cycle on the right-hand side. Whether this cycle can be fully concretised to an execution witness, filling in concrete values in all abstract events, is left as task to a verification back end.

Control flow To build an abstract event graph for branching programs, we consider the if-then-else branches, loops and function calls. Functions are analysed as if they were inlined, thus recursion is not handled. For if-then-else, po in the abstract event graph

[2] The alias analysis we use is known to be sound for the weak architectures we consider [6].

follows both of the branches separately, and then joins at the end of the condition. For loops or backward jumps and given a pair $(x, y) \in \mathsf{po}$, the back-edge may render x reachable from y as well. We thus include copies of x and y in the abstract event graph, such that (y, x) in po if such a back-edge exists. By [7] it suffices to use a single copy, as a critical cycle does not require more than two events in program order per thread.

The analysis proceeds in a forward manner along the control-flow graph of a given program. For each statement recorded in a node of the control-flow graph, the abstract events are computed. When preserved program order is defined via dp (cf. Sec. 2), possible dependencies between abstract events are recorded as well.

4.3 Detecting critical cycles

Given the abstract event graph of a program, we need to compute an over-approximate set of critical cycles. To increase scalability of this procedure, we first identify all strongly connected components (SCCs) in the graph using Tarjan's 1972 algorithm [28], which is linear in the size of the abstract event graph. The detection of critical cycles can then be performed in parallel and independently for each SCC, as no cycle can span multiple SCCs. The SCCs also offer first insights about the program under test: two distinct SCCs will refer to two parts of the code that are independently accessing and updating shared memory.

Detecting all the critical cycles in an SCC Our cycle computation is based on Tarjan's 1973 algorithm [29]. The abstract event graph, however, does not encode the transitive closure of po. Thus, we first extract *candidate cycles* by picking at most two abstract events per thread, which are guaranteed to be (transitively) linked by program order. For each candidate cycle we then perform additional filtering, as such a cycle need not be critical: a candidate is guaranteed to be *not* critical if it does not contain any *unsafe pair* for the given architecture, or is a cycle in *uniproc* or *thin-air*. All of these checks need to be performed a-posteriori for a complete cycle.

Tarjan's original algorithm is worst-case exponential in the number of vertices (abstract events), and our subsequent filtering adds additional complexity. To deal with this complexity, we soundly limit the exploration using properties of critical cycles, such as all program-order pairs per address in a critical cycle being one of write-write, read-write, write-read or read-write-read [4].

4.4 Selecting and Instrumenting Delay Pairs

The above cycle detection yields candidates for unsafe pairs of abstract events to be delayed in each cycle. Following Sec. 3.4, we instrument one pair to delay per cycle. We may select these pairs arbitrarily, but we describe below a weigthed instrumentation which makes the verification time much lower, as we show in Sec. 5.

We first normalise the program such that all shared memory accesses appear in assignments only; any reads in branching conditions or function call parameters are moved to temporary variables as follows: **if** $(\phi(\mathrm{x}))$ ...; $\longmapsto$ tmp = $\phi(\mathrm{x})$; **if**(tmp) ...; for an expression ϕ over a shared memory address x. In the following, we thus restrict ourselves to assignment statements.

For each memory addresses x of events in unsafe pairs x we introduce an array $\mathsf{b}(x)$. In addition to the properties described in Sec. 3.1, we also keep track of the originating thread of the write to x. We introduce an additional pointer for each local variable reading from a shared memory address, i.e. an r such that $r = x;$. In a pair to delay, in one of the critical cycles or after, we equip r with a pointer $\mathsf{q}(r)$, which implements the read queue of Sec. 3.1. We now describe the instrumentation of writes, then reads. To soundly over-approximate all possible behaviours, all instrumented operations are guarded by **if** $(*)$, expressing non-deterministic choice.

Instrumenting writes We implement here the two operations associated to the weak-memory effects of a write w, as defined in Sec. 3.1: (1) delaying a write, $\mathrm{d}(\mathrm{w}(w))$, by appending to the buffer, and (2) flushing a write, $\mathrm{f}(\mathrm{w}(w))$, removing it from the buffer. A delayed write amounts to appending an element to the array:

x = smthg; $\longmapsto$ **if**$(*)$ $\mathsf{b}(x)$.push(smthg,thread.number); **else** x = smthg;

According to Sec. 3.1, each delay is accompanied by a flush. Yet the point in time when the flush happens is not determined. We would thus need to add non-deterministic flush instructions at each statement in the program. This transformation would make the program highly non-deterministic, and very hard for a model checker to analyse. Therefore, we insert flushes only where they might have an effect, i.e. before each potential read from the address that was written to, and make them flush a non-deterministic number of writes in fifo-manner. The function *take* implements the semantics of "write from buffer to memory" of Fig. 3 on C arrays for a non-deterministic number of elements, and returns the resulting in-memory value at address x.

smthg = x; $\longmapsto$ **if**$(*)$ x = $\mathsf{b}(x)$.take(thread.number); smthg = x;

Instrumenting reads Here we are to implement the two operations for reads: enqueuing a delayed read $\mathrm{d}(\mathrm{r}(w, r))$ and reading from the queue, $\mathrm{f}(\mathrm{r}(w, r))$. We delay a read by queueing the memory address to be read from. Note that, given our program normalization, our reads manifest as assignments to local variables. For a local variable r1, we enqueue the read of x as follows:

r1 = x; $\longmapsto$ **if**$(*)$ $\mathsf{q}(r1)$ = &x; **else** r1 = x;

For flushing the read, analogous considerations to the write case are made: we flush non-deterministically upon an actual read (then of r1) only, instead of every program point. The flush dereferences the address previously enqueued:

r2 = r1; $\longmapsto$ **if**($\mathsf{q}(r1)$!= 0 && $*$) { r1 = $*\mathsf{q}(r1)$; $\mathsf{q}(r1)$ = 0;} r2 = r1;

4.5 Weighted selection of unsafe pairs

Above, we selected an arbitrary unsafe pair per cycle, as this suffices to reveal all weak-memory effects (cf. Sec. 3). We do observe, however, that the choice of pairs has a strong effect on verification time. We thus assign an empirically devised cost **d** to candidate pairs. With our implementation, we chose **d**(poW*)=1 (pairs in program

Input: the edges to instrument E, the cycles C_j
Problem: minimise $\sum_{e_i \in E} \mathbf{d}(e_i) * x_i$
s.t. $\forall j, \sum_{e_i \in C_j \cap E} x_i >= 1$ (ensures soundness)
where
e_i is a pair to potentially instrument,
x_i is a Boolean variable stating whether we instrument e_i,
and **d**() is the cost of an instrumentation.
Output: the x_i, stating which pairs to instrument

Figure 9. Mixed integer programming problem to choose the pairs to instrument

order where the first event is a write), **d**(poRW)=2 (read-write pairs in program order), **d**(rfe)=2 (write-read pairs on different threads), **d**(poRR)=3 (read-read pairs in program order). Given a set E of pairs to delay in the graph with critical cycles C_j, we solve the mixed integer programming problem of Fig. 9. Our experiments show that this encoding yields a speedup of 26% over all architectures with an SC bounded model-checker.

5 Experimental Results

We exercised our method and measured its cost using 8 tools. We considered 5 ANSI-C model checkers: a bounded model checker based on CBMC described in [9]; SatAbs, a verifier based on predicate abstraction, using Boom as the model checker for the Boolean program; ESBMC, a bounded model checker; Threader, a thread-modular verifier; and Poirot, which implements a context-bounded translation to sequential programs. These tools cover a broad spectrum of symbolic algorithms for verifying SC programs. We also experimented with Blender, CheckFence, and MMChecker. We ran our experiments on Linux 2.6.32 64-bit machines with 3.07 GHz (only Poirot was run on a Windows system).

Validation First, we systematically validate our setup using 555 litmus tests exposing weak memory artefacts (e.g. instruction reordering, store buffering, write atomicity relaxation) in isolation. The diy tool automatically generates x86, Power and ARM assembly programs implementing an idiom that cannot be reached on SC, but can be reached on a given model. For example, **sb** (Fig. 1) exhibits store buffering, thus the final state can be reached on any weak model, from TSO to Power.

Each litmus test comes with an assertion that models the SC violation exercised by the test, e.g. the outcomes of Fig. 1 and 2. Thus, verifying a litmus test amounts to checking whether the model under scrutiny can reach the specified outcome. We then convert these tests automatically into C code, leading to programs of 48 lines on average, involving 2 to 4 threads.

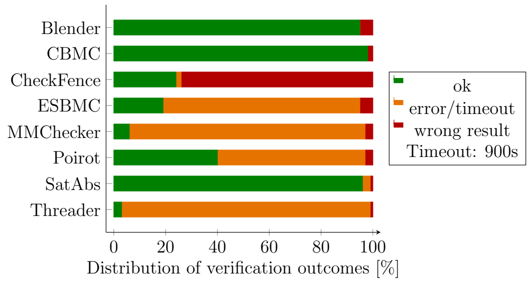
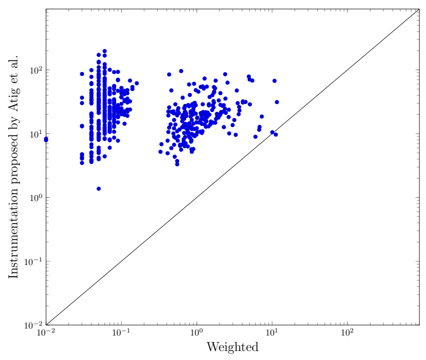
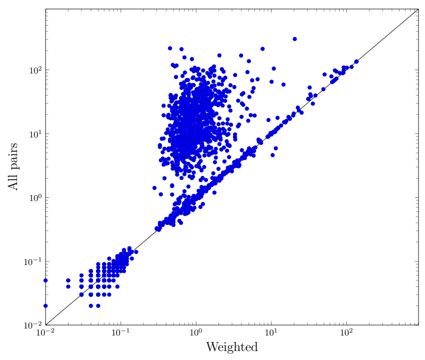


Figure 10. All tools on all litmus tests and models



(a) All accesses [12] vs. weighted selection



(b) All pairs vs. weighted selection

Figure 11. Comparison of verification times (using CBMC) for different instrumentations

These examples provide assurance that we soundly implement the theory of Sec. 3: we verify each test w.r.t. SC, i.e. without transformation, then w.r.t. TSO, PSO, RMO, and Power. Despite the tests being small, they provide challenging concurrent idioms to verify. Fig. 10 compares the tools on all tests and models. Most tools, with the exception of Blender, CBMC and SatAbs, time out or give wrong results on a vast majority of tests. Blender only expectedly fails on tests involving `lwsync` fences; CBMC and SatAbs return spurious results in 1.5% of the tests, caused by the over-approximation in the implementation of instrumentation.

Fig. 11 compares the verification time using CBMC over all litmus families (e.g. rfe tests exercise store atomicity, podwr tests exercise the write-read reordering) for different instrumentation options. First, with the restriction to TSO, Fig. 11(a) compares the instrumentation of all shared memory accesses proposed in [12] to the weighted transformation (Sec. 4.5). On average, we observe a more than 300-fold speedup in verification time. In addition, the reduced instrumentation also yields 246 fewer spurious results. We also quantify the specific benefit of the weighted selection of pairs in Fig. 11(b). We comparing the cost of the instrumentation of all pairs on critical cycles with that of the weighted transformation (Sec. 4.5) for all models, tools and tests. The average speedup over all models and tests is still more than one order of magnitude. We give the detailed results for all experiments online.

We also verified several TSO examples that have been used in the literature (details are online). Note that these examples in fact only exhibit idioms already covered by our litmus tests (e.g. Dekker corresponds to the **sb** test of Fig. 1). Furthermore we applied the instrumentation to code taken from the Read-Copy-Update algorithm in the Linux kernel and scheduling code in the Apache HTTP server, as well as industrial code from IBM. We observe that the instrumentation tool completes even on such code of up to 28,000 lines in less than 1 second, and in 32 seconds on IBM's code. We now study one real-life example in detail, an excerpt of the relational database software PostgreSQL.

Worker Synchronization in PostgreSQL Mid 2011, PostgreSQL developers observed that a regression test occasionally failed on a multi-core PowerPC system.[3] The test implements a protocol passing a token in a ring of processes. Further analysis drew the attention to an interprocess signalling mechanism. It turned out that the code had already been subject to an inconclusive discussion in late 2010.[4]

```
1  #define WORKERS 2
2  volatile _Bool latch [WORKERS];
3  volatile _Bool flag [WORKERS];
4  void worker(int i )
5  { while(! latch [ i ]);
6    for (;;)
7    { assert (! latch [ i ] || flag [ i ]);
8      latch [ i ] = 0;
9      if ( flag [ i ])
10     { flag [ i ] = 0;
11       flag [( i+1)%WORKERS] = 1;
12       latch [( i+1)%WORKERS] = 1; }
13     while(! latch [ i ]);   } }
```

Listing 1.1. Token passing in pgsql.c

The code in Listing 1.1 is an inlined version of the problematic code, with an additional assertion in line 7. Each element of the array "latch" is a Boolean variable stored in shared memory to facilitate interprocess communication. Each working process waits to have its latch set and then expects to have work to do (from line 9 onwards). Here, the work consists of passing around a token *via* the array "flag". Once the process is done with its work, it passes the token on (line 11), and sets the latch of the process the token was passed to (line 12).

Starvation seemingly cannot occur: when a process is woken up, it has work to do (has the token). Yet, the PostgreSQL developers observed that the wait in line 13 (which in the original code is bounded in time) would time out, thus signalling starvation of the ring of processes. The developers identified the memory model of the platform as possible culprit: it was assumed that the processor would at times delay the write in line 11 until after the latch had been set.

We transform the code of Listing 1.1 for two workers under Power. The event graphs show two idioms: **lb** (load buffering) and **mp** (message passing), in Fig. 12 and 13. The code fragments on the left-hand side give the corresponding line numbers in Listing 1.1.

The **lb** idiom contains the two *if* statements controlling the access to both critical sections. Since the **lb** idiom is yet unimplemented by Power machines (despite being allowed by the architecture [27]), we believe that this is not the bug observed by the PostgreSQL developers. Yet, it might lead to actual bugs on future machines.

In contrast, the **mp** case is commonly observed on Power machines (e.g. 1.7G/167G on Power 7 [27]). The **mp** case arises in the PostgreSQL code by the combination of some writes in the critical section of the first worker, and the access to the critical section of the second worker; the relevant code lines are in Fig. 13.

We first check the fully transformed code with SatAbs. After 21.34 seconds, SatAbs provides a counterexample (given online), where we first execute the first worker up to line 17. All accesses are w.r.t. memory, except at lines 14 and 15, where the values 0 and 1 are stored into the buffers of flag[0] and flag[1]. Then the second worker starts, reading the updated value 1 of latch[1]. It exits the blocking while (line 7) and reaches the assertion. Here, latch[1] still holds 1, and flag[1] still holds 0, as Worker 0 has not yet flushed the write waiting in its buffer. Thus, the condition of the *if* is not true, the

[3] http://archives.postgresql.org/pgsql-hackers/2011-08/msg00330.php
[4] http://archives.postgresql.org/pgsql-hackers/2010-11/msg01575.php

pgsql (lb)

Worker 0	Worker 1
(12) `if(flag[0])`	(12) `if(flag[1])`
(15) `flag[1]=1;`	(15) `flag[0]=1;`

Observed: `flag[0]=1; flag[1]=1`

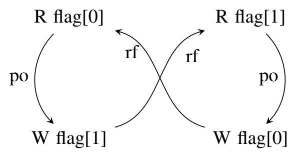


Figure 12. An **lb** idiom detected in `pgsql.c`

pgsql (mp)

Worker 0	Worker 1
(15) `flag[1]=1;`	(7) `while(!latch[1]);`
(16) `latch[1]=1;`	(12) `if(flag[1])`

Observed: `latch[1]=1; flag[1]=0`

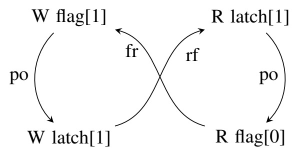


Figure 13. An **mp** idiom detected in `pgsql.c`

critical section is skipped, and the program arrives line 19, without having authorised the next worker to enter the critical section, and loops forever.

As **mp** can arise on Power e.g. because of non-atomic writes, we know by Sec. 3.4 that we only need to transform one rfe pair of the cycle, and relaunch the verification. SatAbs spends 1.29 seconds to check it (and finds a counterexample, as previously).

PostgreSQL developers discussed fixes, but only committed comments to the code base, as it remained unclear whether the intended fixes were appropriate. We proposed a provably correct patch solving both **lb** and **mp**. After discussion with the developers[5], we improved it to meet the developers' desire to maintain the current API. The final patch introduces two `lwsync` barriers: after line 8 and before line 12.

6 Conclusion

We presented a provably sound method to verify concurrent software w.r.t. weak memory. Our contribution allows to lift SC methods and tools to a wide range of weak memory models (from x86 to Power), by means of program transformation.

Our approach crucially relies on the definition of a generic operational model equivalent to the axiomatic one of [8]. We do not favor any style of model in particular, but we highlight the importance of the availability of several equivalent mathematical styles to model semantics as intricate as weak memory. In addition, operational models are often the style of choice in the verification community; we contribute here to the vocabulary to tackle the verification problem w.r.t. weak memory.

Our extensive experiments and in particular the PostgreSQL bug demonstrate the practicability of our approach from several different perspectives. First, we confirmed a known bug (**mp**), and validated the fix proposed by the developers, including an evaluation of different synchronisation options. Second, we found an additional idiom (**lb**), which will be a bug on future Power machines; our fix repairs it already.

References

1. http://research.microsoft.com/en-us/projects/poirot

[5] http://archives.postgresql.org/pgsql-hackers/2012-03/msg01506.php

2. Abdulla, P., Atig, M.F., Chen, Y., Leonardsson, C., Rezine, A.: Counter-Example Guided Fence Insertion under TSO. In: TACAS (2012)
3. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29, 66–76 (1995)
4. Alglave, J.: A Shared Memory Poetics. Ph.D. thesis, Université Paris 7 and INRIA (2010)
5. Alglave, J.: A Formal Hierarchy of Weak Memory Models. In: FMSD (2012)
6. Alglave, J., Kroening, D., Lugton, J., Nimal, V., Tautschnig, M.: Soundness of data flow analyses for weak memory models. In: APLAS (2011)
7. Alglave, J., Maranget, L.: Stability in weak memory models. In: CAV (2011)
8. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models. In: CAV (2010)
9. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient BMC of concurrent software, submitted to TACAS 2013, available at http://www.cprover.org/etaps/
10. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL (2010)
11. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: ESOP (2012)
12. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in the analysis of weak memory models. In: CAV (2011)
13. Bouajjani, A., Meyer, R., Moehlmann, E.: Deciding robustness against total store ordering. In: ICALP (2011)
14. Burckhardt, S., Alur, R., Martin, M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
15. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE. pp. 331–340. ACM (2011)
16. Donaldson, A., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: CAV (2011)
17. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. In: CAV (2011)
18. Huynh, T., Roychoudhury, A.: A memory model sensitive checker for C#. In: FM (2006)
19. Jin, H., Yavuz-Kahveci, T., Sanders, B.A.: Java memory model-aware model checking. In: TACAS (2012)
20. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
21. Kuperstein, M., Vechev, M., Yahav, E.: Partial-Coherence Abstractions for Relaxed Memory Models. In: PLDI (2011)
22. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1979)
23. Linden, A., P.Wolper: A verification-based approach to memory fence insertion in relaxed memory systems. In: SPIN (2011)
24. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: ECOOP (2010)
25. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOL (2009)
26. Park, S., Dill, D.: An executable specification, analyzer and verifier for RMO. In: SPAA (1995)
27. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding Power multiprocessors. In: PLDI (2011)
28. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. (1972)
29. Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM J. Comput. (1973)
30. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Memory model sensitive data race analysis. In: ICFEM (2004)