Software Verification for Weak Memory via Program Transformation*

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

Department of Computer Science, University of Oxford, UK

Abstract Despite multiprocessors implementing weak memory models, verification methods often assume *Sequential Consistency* (SC), 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/ARM) 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 [20]. 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 [27], hence might miss bugs specific to weak memory. Recent work addresses the design or the adaptation of existing methods and tools to weak memory [25,31,15,9,23,8,2], 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) [6]. In practice, this means that we cannot design a *complete* verification method. Yet, we can

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

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.

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. Sec. 3 details our *transformation* for concurrent programs on weak memory. This requires defining a generic abstract machine that we prove (in the Coq proof assistant) equivalent to the framework of [5] (recalled in Sec. 2). Sec. 3 shows a drastic optimisation of the transformation, and we prove that this is sound.
2. Sec. 4 describes our implementation, where the generality of our approach reveals itself the most: we support a broad variety of models (x86/TSO, PSO, RMO and Power) and program analysers (Blender [18], CheckFence [9], ESBMC [11], MMChecker [15], Poirot [1], SatAbs [13], Threader [14], and our new CImpact tool, an extension of Impact [22] to SC concurrency).
3. Sec. 5 details our experiments using this setup. i) We systematically validate our implementation w.r.t. our theoretical study with 555 *litmus tests*, generated by the diy tool [5] to exercise weak memory artefacts in isolation. ii) We verify several TSO examples from the literature [12,26,19,30,10]. iii) We verify a new example, which is an excerpt of the relational database software PostgreSQL and has a bug specific to Power. This bug raised notable interest at IBM, and we are already trying our tools on their software.

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

Related Work We focus here on the *verification* problem, i.e. forbidding the behaviours that are buggy, not all the non-SC ones. This problem is non-primitive recursive for TSO [6]. It is undecidable if the reads are smart (i.e. they can guess the value that they will read eventually), e.g. for RMO-like models [6]. Forbidding *causal loops* restores decidability; relaxing write atomicity makes the problem undecidable again [7].

Previous work therefore compromise by choosing various bounds over the objects of the model [8,17], over-approximating the possible behaviours [18,16], or relinquishing termination [21]. For TSO, [2] presents a sound and complete solution.

By contrast, we disregard in the present paper any completeness issue. We are not primarily concerned with efficiency either, although we do provide a drastic optimisation of our transformation. focus in this work on the soundness, generality, and implementability of our method, to bridge the gap between theory and practice. We emphasise the fact that our method allows to lift any SC method or tool to a large spectrum of weak memory models, ranging from x86 to Power.

2 Context: Axiomatic Model

We use the framework of [5], which provably embraces several *architectures*: SC [20], Sun TSO (i.e. the x86 model [24]), PSO and RMO, Alpha, and a fragment of Power. We present this framework via *litmus tests*, as shown in Fig. 1.

sb

P_0	P_1
(a) `x` ← `1`	(c) `y` ← `1`
(b) `r1` ← `y`	(d) `r2` ← `x`

Allowed? `r1=0`; `r2=0`

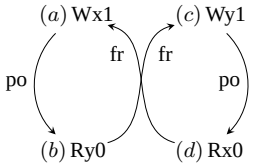


Figure 1. Store Buffering (**sb**)

iriw

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

Allowed? `r1=1`; `r2=0`; `r3=1`; `r4=0`;

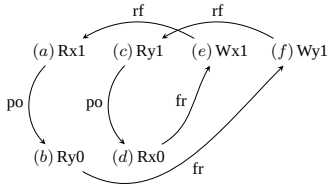


Figure 2. Independent Reads of Independent Writes (**iriw**)

The keyword *allowed* asks if a given architecture allows the outcome "`r1=0;r2=0`". This relates to the execution graphs of this program, composed of relations over *read and write memory events*. A store instruction (e.g. `x` ← `1` on P_0) corresponds to a write event ((a)Wx1), and a load (e.g. `r1` ← `y` on P_0) to a read ((b)Ry0). The validity of an execution boils down to the absence of certain cycles in the execution graph. Indeed, 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.

If an execution graph has a cycle, we check if the architecture *relaxes* some relations in this cycle. The consensus can ignore a relaxed relation, hence become acyclic, i.e. the architecture allows the final state. In Fig. 1, on SC where no relation is relaxed, the cycle forbids the execution. x86 relaxes the program order (po in Fig. 1) between writes and reads, thus a forbidding cycle no longer exists since (a, b) and (c, d) are relaxed.

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 to the event $(e)\mathrm{W}x1$.

We associate the program with 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) modelling *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.

Then, 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 and register initially holds 0. If `r1=1` in the end, the read (a) read its value from the write (e) on P_2, hence $(e, a) \in \mathsf{rf}$. If `r2=0` in the end, the read (b) read 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 when the write w transits in store buffers and caches, a processor can read its value through a read r 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. Architectures provide special *fence* (or *barrier*) instructions, to prevent weak behaviours. Following [5], the relation fence $\subseteq$ 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 rfe; fence (resp. fence; 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 [5], we consider the write serialisation ws and the from-read relation fr to be always *safe*, i.e. not relaxed. 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 are 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 po-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

We want to transform a program P into a program P' so that executing P' on SC gives us the behaviours that P exhibits on a weak architecture. To do so, we define an *abstract machine*, composed of a store buffer per address and a load queue. We avoid defining one machine per architecture as follows. We first define a *core machine*, implementing the $\mathrm{uniproc}$ and thin checks common to all models. Then, we implement an architecture A by adding, on top of the core, a *protocol* ordering the entering and exiting the buffers and queue. This protocol enforces the consensus order defined by A.

3.1 The Abstract Machine

A *state* s is either $\bot$ or a tuple $(\mathsf{m}, \mathsf{b}, \mathsf{q}, \mathsf{l})$, which contains (writing **addr**, **evt**, **proc**, **rln** for the types of memory addresses, events, processors (or thread ids) and relations):

the *memory* m : **addr** $\to$ **evt** maps a memory address ℓ to a write to ℓ;
a *buffer* b : **addr** $\to$ **rln evt** a total order over writes to the same address;
a *queue* q : **rln (evt** $\times$ **evt)** over reads, tracking instruction dependencies;
the *log* l : **proc** $\times$ **addr** $\to$ **evt** provides the last write to address ℓ seen by thread p.

Note that our buffer is not a per-thread object, but solely a per-location object, as opposed to most existing formalisations. 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.

Core machine We modify a state *via labels*. Given an execution (E, X), we define our labels from the events of E. First, we *augment* our events: a write w becomes $\mathrm{w}(w)$ and r becomes $\mathrm{r}(w, r)$, where w is the write from which r reads (i.e. $(w, r) \in \mathsf{rf}$). Then we tag an augmented event e to build the labels $\mathrm{d}(e)$ and $\mathrm{f}(e)$. In effect, we *split* an event e into its *delayed* part $\mathrm{d}(e)$ (the part entering the buffer or the queue), and its *flushed* part $\mathrm{f}(e)$ (the part exiting the buffer or the queue).

A label modifies a state $\mathsf{s} = (\mathsf{m}, \mathsf{b}, \mathsf{q}, \mathsf{l})$ as follows. We describe the transitions in prose, and omit their formal definitions for brevity (our Coq proofs are available online):

Write to buffer a write $\mathrm{d}(\mathrm{w}(w))$ to location ℓ can always enter the buffer b, taking its place after all the writes to ℓ that are already in b;
Write from buffer to memory a write $\mathrm{f}(\mathrm{w}(w))$ to location ℓ and from thread p exits the buffer b and updates the memory to ℓ if there is no write to ℓ pending in b, and if there is no pending read from ℓ enqueued by p before w entered b;
Enqueue read a read $\mathrm{d}(\mathrm{r}(w, r))$ can always enter the queue q, taking its place after all the reads in q on which r depends (i.e. $\{r' \mid (r', r) \in \mathsf{dp} \vee (r', w) \in \mathsf{dp}\}$);
Read from queue a read $\mathrm{f}(\mathrm{r}(w, r))$ by thread p from ℓ exits the queue and updates the log relative to p and ℓ if there is no pending read on which r depends; if no write to ℓ and from p entered the buffer b after (resp. before) r yet took its place in b before (resp. after) w; and if w took its place in b after the last write to ℓ seen by p.

The rules for writes enforce the existence of a write serialisation, since we order the writes to a given location in the buffer, then flush them in the same order. Reads are smart (to the extent that they respect $\mathrm{uniproc}$ or thin): we flush a read $\mathrm{f}(\mathrm{r}(w, r))$ if the write w lies in the memory or in the buffer, i.e. in any level of the memory hierarchy.

Finally, note that the core machine implements the $\mathrm{uniproc}$ and thin checks that all architectures satisfy, thanks to the premises to exiting the buffer or the queue.

Consensus protocol Now, to implement a particular architecture A, the machine also needs to implement the consensus defined by A. We do so by defining a protocol that constrains the order in which reads and writes exit the queue and the buffer, as follows.

We first gather the *delay pairs* of A (adapting the terminology of [29,4]). The delays make a program behave differently on A than on SC, as follows [5,4]. First, when the program has a relaxed program order pair, e.g. (a, b) in Fig. 1 on TSO. Second, when the program reads from a non-atomic write, e.g. (e, a) in Fig. 2 on Power. Formally:

$$\mathrm{delays}_A(E, X) \triangleq \{(e, e') \in (\mathsf{po} \cup \mathsf{rfe}) \wedge (e, e') \notin \mathrm{safe}_A\}$$

Note that there is no delay on SC, and that the rfe case only concerns architectures relaxing write atomicity, e.g. Power/ARM, but not x86. In practice, the delays are: the write-read pairs on x86/TSO, e.g. in Fig. 1 (a, b) and (c, d); the write-read and write-write pairs on PSO; all po pairs (except dp ones) on RMO and Power; all rfe pairs, on Power, e.g. in Fig. 2 (e, a) and (f, c).

In the following, e stands for both the event e and its augmented event. To implement the consensus order defined by a given architecture A, we *augment our core machine with an A-protocol*. Formally, we feed it a path of labels $\mathrm{path}(E, X, \mathbb{D})$ defined as follows (where $\mathbb{D}$ is a set of pairs to be delayed):

Enter in po we ensure that our machine has an SC semantics, by forcing two events e_1 and e_2 in program order to enter the buffer and the queue in this order; i.e. $(e_1, e_2) \in \mathsf{po} \Rightarrow (\mathrm{d}(e_1), \mathrm{d}(e_2)) \in \mathrm{path}(E, X, \mathbb{D})$;

Safe exit (se) if (e_1, e_2) does not form a delay, we force them to exit the buffer and the queue in the same order i.e. $(e_1, e_2) \notin \mathbb{D} \Rightarrow (\mathrm{f}(e_1), \mathrm{f}(e_2)) \in \mathrm{path}(E, X, \mathbb{D})$;

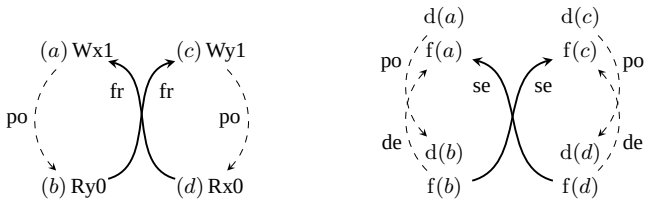
Delayed exit (de) if (e_1, e_2) forms a delay, we force them to exit the buffer and the queue in the converse order i.e. $(e_1, e_2) \in \mathbb{D} \Rightarrow (\mathrm{f}(e_2), \mathrm{f}(e_1)) \in \mathrm{path}(E, X, \mathbb{D})$.

Observe that if there are no delay pairs (i.e. $\mathbb{D} = \emptyset$), this definition describes all possible interleavings, i.e. our machine implements SC:

Lemma 1. $\mathrm{valid}_{SC}(E, X) \Leftrightarrow \mathrm{mns}(E, \mathrm{path}(E, X, \emptyset))$

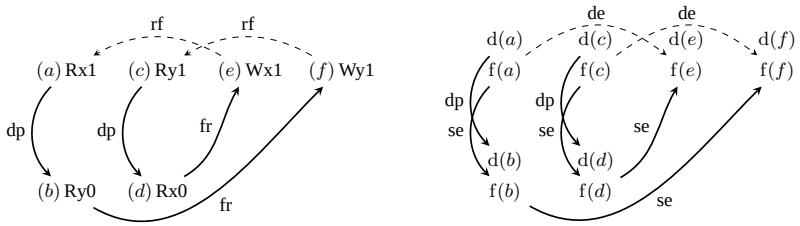
Examples We illustrate how the machine implements TSO or Power by revisiting the **sb** test of Fig. 1 for TSO and the **iriw** test of Fig. 2 for Power.

In Fig. 1, the pairs (a, b) on P_0 and (c, d) on P_1 are delays on TSO. Our machine simulates the weak behaviour exhibited on TSO, following the scenario in Fig. 3. The machine buffers or enqueues all events w.r.t. program order. Since (a, b) and (c, d) are delays on TSO, the machine augmented with a TSO protocol flushes the reads b and d before the writes a and c, ensuring that the registers `r1` and `r2` hold 0 in the end.



(a) Axiomatic model (b) Machine model

Figure 3. Revisiting **sb** with our core machine augmented with a TSO protocol



(a) Axiomatic model (b) Machine model

Figure 4. Revisiting **iriw+dps** with our core machine augmented with a Power protocol

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) are delays, because Power has non-atomic writes. Our machine simulates the weak behaviour exhibited on Power, following Fig. 4. The machine enqueues or buffers all events w.r.t. program order. Since (a, b) and (c, d) are safe on Power, our machine augmented with a Power protocol flushes a before b (resp. c before d). The writes corresponding to a and c are in the buffer, ensuring that `r1` and `r3` hold the value 1 in the end. 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, since (e, a) and (f, c) are delays, the machine flushes them in the converse order.

Formally, we establish the equivalence of the axiomatic definition of A (cf. Sec. 2) with our machine augmented with an A-protocol. We write mns for *machine not stuck*, i.e. the machine cannot relate any state to $\bot$ when reading a given path of labels:

Thm. 1. $\mathrm{valid}_A(E, X) \wedge \neg(\mathrm{valid}_{SC}(E, X)) \Leftrightarrow \mathrm{mns}(E, \mathrm{path}(E, X, \mathrm{delays}_A(E, X)))$

Thm. 1 shows that our machine provides a hierarchy of weak memory models equivalent to the one of [5], but in an operational style rather than an axiomatic one.

Let us explain what intuitively matters for Thm. 1 to hold, or in other words for us to be able to simulate a weak execution in an SC world. A weak execution will contain at least one cycle that contradicts the definition of SC, e.g. in Fig. 1 and 2. Such a cycle contradicts SC in that it violates program order: for example in Fig. 1, the pair (a, b) on P_0 is in po, but there is also a path from b to a.

To enable SC reasoning, we need to dismantle the cycle, as illustrated in Fig. 3 and Fig. 4. These figures recall on the left the axiomatic cycles of Fig. 1 and 2. On the

right, they show the machine counterparts of the axiomatic cycles. We use the following graphical conventions. In the axiomatic world (i.e. on the left of our figures), we reflect a delay pair by a dashed arrow. For example in the **sb** test of Fig. 3 on TSO, the write-read pairs (a, b) and (d, c) are delayed. In the **iriw+dps** test of Fig. 4 on Power, the read-from pairs (e, a) and (f, c) are delayed (as opposed to the read-read pairs (a, b) on P_0 and (c, d) on P_1, which are safe thanks to the dependencies). In the machine world, the Delayed exit rule (i.e. the machine counterpart of delayed pairs) is depicted with a dashed arrow. For safe pairs and the safe exit rule (the machine counterpart of safe pairs), we use thick arrows, e.g. the dependency dp between a and b on P_0 in **iriw+dps**.

First, we dismantle the axiomatic cycle by splitting an event into its delay and flush parts, e.g. the write a on P_0 in **sb** becomes $\mathrm{d}(a)$ and $\mathrm{f}(a)$. Then, we enable SC reasoning by enforcing consistency with the program order. For example in Fig. 3, the pair (a, b) is in po on the left, which we reflect on the right by pushing $\mathrm{d}(a)$ and $\mathrm{d}(b)$ in the buffer and the queue in this order, as depicted by the po arrow.

Then, the Delayed exit rule de allows us to create *a diversion* from the cycle: we flush first $\mathrm{f}(b)$ then $\mathrm{f}(a)$ as depicted by the de arrow between $\mathrm{f}(b)$ and $\mathrm{f}(a)$ on the right.

Similarly for the **iriw+dps** example recalled on the left of Fig. 4, we split all events into their d and f parts, then create a diversion from the axiomatic cycle by using the Delayed exit rule on e.g. the (e, a) pair, as depicted by the de arrow between $\mathrm{f}(a)$ and $\mathrm{f}(e)$ on the right. This means that we flush the read $\mathrm{f}(a)$ before the write $\mathrm{f}(e)$, hence the read a occurs from the queue, i.e. a reads the value of e from the buffer.

3.2 Reducing the Number of Delay Pairs

Crucially, the notion of creating a diversion from an SC cycle allows us to optimise the number of pairs that we delay. Lem. 1 shows that when (E, X) is SC (despite the program being run on a weak architecture A), no pair needs delaying.

Critical delays Consider (E, X) valid on A but not on SC, as in Fig. 1. The pairs (a, b) and (c, d) are delays on TSO, i.e. both *might* be delayed. But it suffices to delay one of them to reveal the weak behaviour. Indeed, it is sufficient to buffer e.g. the write (a) to x on P_0, perform all the other events from memory (thus $\mathtt{r1} = 0$ and $\mathtt{r2} = 0$ because the write (a) lies in the buffer), and finally flush the write (a) to memory.

In the non-SC execution of Fig. 2, (e, a), (f, c) are delays on Power (assuming dependencies to make (a, b) and (c, d) safe), but it suffices to delay e.g. (e, a). We buffer the write (e), enqueue the read (a), and perform other events from memory. This corresponds to a caching scenario where P_0 and P_3 communicate privately *via* x (since they communicate from the buffer to the queue), all the other communications occurring from memory, in an SC fashion.

Note that in both cases, the reasoning would have been similar if we had chosen another delay pair along the cycle exhibited by the execution, or if we had delayed more events than just the ones we chose. As said before, what matters is to delay enough pairs to form a diversion from the cycle, to enable an SC reasoning.

We said before that when an execution is SC, no pair needs delaying. [4, Thm.1] characterises the non-SC executions by the presence of certain cycles, called *critical cycles*, which satisfy the two following conditions. **(i)** Per processor, the cycle involves

at most two memory accesses a and b on this processor and $\mathrm{addr}(a) \neq \mathrm{addr}(b)$. **(ii)** For a given memory location x, the cycle involves at most three accesses relative to x, and these accesses are from distinct processors. The executions of the tests **sb** and **iriw** in Fig. 1 and 2 give typical examples of critical cycles.

Thus, if there is a weak memory specific bug in an execution, it is along a critical cycle, or on the remainder of a path after a critical cycle. Formally, we show that we only need to delay (i.e. apply the Delayed exit rule de) *one* delay pair per critical cycle to simulate a weak execution with our machine (writing $\mathrm{crits}_A(E, X)$ for *any* selection of pairs in $\mathrm{delays}_A(E, X)$ with at least one pair per critical cycle of E):

Thm. 2. $(\mathrm{valid}_A(E, X) \wedge \neg(\mathrm{valid}_{SC}(E, X))) \Leftrightarrow \mathrm{mns}(E, \mathrm{path}(E, X, \mathrm{crits}_A(E, X)))$

Thm. 2 has several consequences. From the semantic perspective, it defines a family of paths equivalent to (E, X), i.e. the paths built inductively as above from any selection of critical pairs. Thus one can see the partial orders given by (E, X) in the axiomatic model as the canonical representation of all the equivalent operational paths.

From the verification perspective, the critical pairs highlight the instructions that should be delayed when verifying a program running on weak memory. Crucially, we show that only one delay pair per critical cycle actually needs delaying. This enables efficient verification, as shown by our experiments (cf. in Sec. 5 the time taken by SatAbs to verify the two versions of the PostgreSQL excerpt: 21.34 vs. 1.29 seconds).

Transformation Consider a non-SC execution and a selection of delays, e.g. the execution of **sb** in Fig. 3 and the pair (a, b). As said before, we only need to create a diversion from the cycle, here by using the Delay exit rule de on (a, b), i.e. flushing $\mathrm{f}(b)$ then $\mathrm{f}(a)$ as depicted on the right of Fig. 3. Consider now that the other pairs, in our example (c, d), are not delayed, in the sense that we use the Safe exit rule to handle them; this amounts to having an se arrow between $\mathrm{f}(c)$ and $\mathrm{f}(d)$. This scenario corresponds to a situation where the write c writes directly to memory (i.e. in our machine writes to the buffer but is flushed immediately after), and the read d reads from memory as well.

Thus in practice, we will tag an event with m to mean that we perform it w.r.t. memory. Otherwise, given a selection $\mathbb{D}$ of delays, we delay an event e (i.e. tag it d) when:

Source of program order (dpo) there is an event e' after e in program order (i.e. $(e, e') \in$ po), forming a delay pair with e (i.e. $(e, e') \in \mathbb{D}$); or

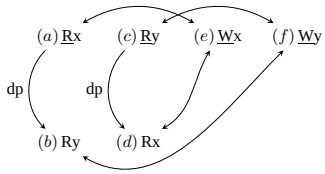
Source of read-from (drfs) there is a read e' from another thread reading from e (i.e. $(e, e') \in$ rfe), forming a delay pair with e (i.e. $(e, e') \in \mathbb{D}$); or

Target of read-from (drft) there is a write e' from another thread from which e reads (i.e. $(e', e) \in$ rfe), forming a delay pair with e (i.e. $(e', e) \in \mathbb{D}$).

Thus, we simulate a non-SC execution on an architecture A as follows: 1. We find the critical cycles. 2. We select at least one delay pair per critical cycle. 3. We tag the events in these pairs with d w.r.t. **(dpo)**, **(drfs)** and **(drft)**, and all the others with m. 4. We perform the events tagged d from the buffer or the queue, and the events tagged m from memory. 5. We flush the delay pairs following the Delayed exit rule de, and the other pairs following the Safe exit rule se.

```
1 thd1:          thd2:          thd3:   thd4:
2 r1:=x          r3:=y          x:=1    y:=1
3 tmp1:=x xor x  tmp2:=y xor y
4 r2:=y+tmp1     r4:=x+tmp2

6 main:
7 thd1() || thd2() || thd3() || thd4()
8 assert (!( r1=1 & r2=0 & r3=1 & r4=0))
```




```
1 thd1:                          thd3:                          main:
2 // was r1:=x                   // was x:=1                    [...]
3 if (∗)                         if (∗)                         // was assert
4   r1:= buff_x . take (thd1)      buff_x . push (1, thd3)      if (!( delay_r1 =0) & ∗)
5 else                           else                             r1:= deref ( delay_r1 )
6   delay_r1 := ref (x) end        x:=1 end                       delay_r1 :=0 end
7 [...]                                                         // same for delay_r2
8                                                               assert (!( r1=1 & r2=0
9 // same for thd2               // same for thd4                        & r3=1 & r4=0))
```

Figure 5. Study of transformation of **iriw+dps**

4 Implementation

We implemented the transformation technique described above. Our tool reads a concurrent C program, and generates a new concurrent C program augmented with buffers and queues, which is then passed to an SC verification tool. We added two memory fences (`fence` and `lwfence`) as new C keywords to support x86's `mfence` and Power's `sync` and `lwsync` with the semantics presented in Sec. 2.

We first translate the C source code into a *goto-program* (a control flow graph), then feed it to goto-instrument (a tool automating transformations of goto-programs). We have extended goto-instrument with the transformation described in Sec. 3: given a memory model (x86/TSO, PSO, RMO and Power are available for now), goto-instrument adds the instructions necessary to transform the delay pairs.

Let us explain how we transform a program using the example **iriw+dps** (a variation of the **iriw** test of Fig. 2, augmented with dependencies between the read pairs to make these pairs safe). We give this program (in C) at the top of Fig. 5.

To transform **iriw+dps**, goto-instrument first produces the abstract event graph on the right-hand side of Fig. 5 from the control-flow graph of the program, which over-approximates the event structures by ignoring the values of the variables, and each abstract event may correspond to an unbounded number of concrete events. The tool then computes possible critical cycles on abstract events. By taking into account any possible concretisation of these events, these cycles are an over-approximation of the actual cycles present in concrete executions of this program, but spurious cycles do not impair the soundness of our method. On our example, due to the direction of dp arrows, the only possible cycle corresponds to the one in Fig. 2: a, b, f, c, d, e, a.

Next, goto-instrument computes the delay pairs for the selected memory model (here, for Power, (e, a), and (f, c)). Following Sec. 3.2, we can transform *all* the delay

pairs (following Thm. 1), or only one pair per cycle (following Thm. 2). As we discuss in Sec. 5, this choice can have a drastic impact on the time required by the program analyser (e.g. 21.34 vs. 1.29 seconds for SatAbs on the PostgreSQL excerpt).

The tool then transforms these pairs (underlined instructions and events in Fig. 5) using a buffer of size 2 for each variable. To ensure soundness despite this limitation, it adds assertions to check whether this buffer bound is exceeded.

The lower part of Fig. 5 gives an excerpt of the resulting transformed program (only the first read of `thd1`, and the write of `thd3`). We transform a write into `x` (e.g. `x:=1` in `thd3`) with a non-deterministic choice (`if(*)`): either the write directly hits the memory, as it would without transformation, or the write is stored into the buffer (`buff_x.push(1,thd3)`). We transform a load from `x` into register `r1` (e.g. `r1:=x` in `thd1`) with another non-deterministic choice: either a load from the memory or from the write buffer of `x` (`r1:=buff_x.take(thd1)`), or a postponed read (`delay_r1:=ref(x)`).

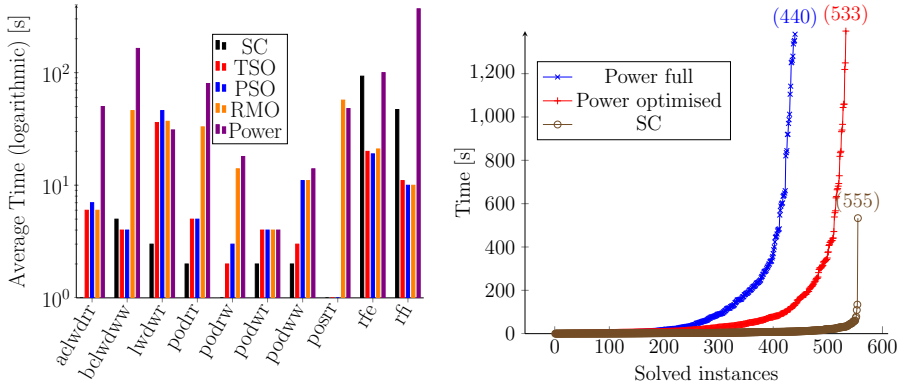
Postponing a read models the read entering the queue. In this case, we do not update `r1`, but the pointer (initially null) `delay_r1` is set to `x` (`delay_r1:=ref(x)` where `ref(x)` returns the address of `x`). This pointer 1) states that the value of `r1` might be affected by a delayed read in the queue (as it is not null anymore), and 2) keeps track of the variable `x`, which can be read by a read in the queue. When a subsequent instruction reads the value of `r1` (e.g. `assert(!(r1 & ...))` in function `main`) and `delay_r1` is null, then we read the current value of `r1` because there is no delayed read in the queue affecting `r1`. If `delay_r1` points to a variable `x`, then there are two cases (again a non-deterministic choice). Either the flush of the read of `x` in the queue happens after the `assert`, in which case `assert` reads the current value of `r1`. Or the flush happens before, in which case we update `r1` with the current value in memory of the variable pointed to by `delay_r1`, namely `x` (*via* `r1:=deref(delay_r1)`, where `deref` dereferences the pointer in argument).

The transformed goto-program can be given directly to CImpact or SatAbs. Alternatively, we can convert it back into C code, and hand it to any program analyser that can read C source (e.g. CheckFence, ESBMC, Poirot, Threader). We also wrote a converter to Blender's input format, and a C# generator for MMChecker.

5 Experimental Results

We exercised our method and measured its cost using 8 tools. We considered 5 ANSI-C model checkers: SatAbs, a verifier based on predicate abstraction, using Boom as the model checker for the Boolean program; ESBMC, a bounded model checker; CImpact, a variant of the Impact algorithm extended to SC concurrency; 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



(a) Average of SatAbs on selected litmus tests (b) SatAbs on SC, Power full and optimised

Figure 6. Selected experimental results

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, the **sb** test of Fig. 1 exhibits store buffering, thus can be reached on any weak model, from TSO to Power. The **iriw** test of Fig. 2 can only be reached on RMO (by reordering the reads) or on Power (for the same reason, or because the writes are non-atomic). Finally, **iriw+dps** (i.e. **iriw** with dependencies between the reads to prevent their reordering) can only be reached on Power.

Each litmus test comes with an assertion expressing 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.

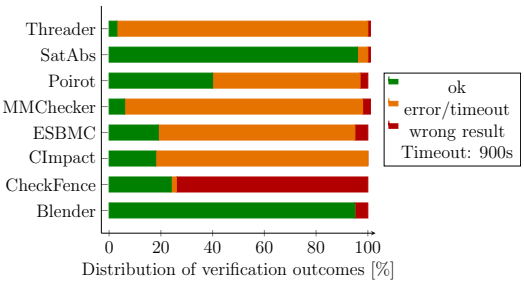


Figure 7. Comparison of tools on all tests and models

These examples allow us to check that we soundly implement the theory of Sec. 3: we verify each test w.r.t. SC, i.e. no transformation, then w.r.t. TSO, PSO, RMO, and Power. Despite the tests being small, they provide challenging concurrent idioms to verify. Fig. 7 compares the tools on all tests and models. Most tools, with the exception of Blender and SatAbs, timeout or give wrong results on a vast majority of tests.

Fig. 6(a) gives the average time that SatAbs needs to verify several litmus families (e.g. rfe tests exercise store atomicity, podwr tests exercise the write-read reordering), for all tools and models, from SC to Power. We also compare the full (Sec. 3.1) and optimised transformation (Sec. 3.2) for all models, tools and tests. Thus each run consists of 9×555 distinct instances. Fig. 6(b) shows that the optimised approach allows SatAbs

to verify 533 of the 555 tests, and 440 with the full approach, in more than twice the time needed for SC. We give the results for all experiments online.

We also verified several TSO examples (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). We now study a real-life example, 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.[1] 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.[2]

```
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. C source code of token passing

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. Manual inspection 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 goto-instrument graph shows two idioms: **lb** (load buffering) and **mp** (message passing), in Fig. 8 and 9, specifying the corresponding lines of 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 [28]), we believe that this is not the bug observed by the PostgreSQL developers. Yet, it might lead to actual bugs on future machines.

By contrast, the **mp** case is commonly observed on Power machines (e.g. 1.7G/167G on Power 7 [28]). 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, which lines we give in Fig. 9.

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

pgsql (lb)

Worker 0	Worker 1
$(0:12)$ `if(flag[0])`	$(1:12)$ `if(flag[1])`
$(0:15)$ `flag[1]=1;`	$(1:15)$ `flag[0]=1;`
Observed! `flag[0]=1; flag[1]=1`	

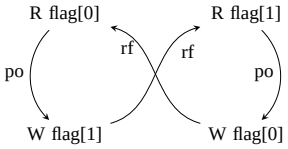


Figure 8. An **lb** idiom detected in `pgsql.c`

pgsql (mp)

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

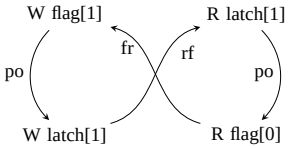


Figure 9. An **mp** idiom detected in `pgsql.c`

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 the 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 flushed yet the write waiting in its buffer. Thus, the condition of the *if* is not true, the critical section is skipped, and the program arrives line 19, without having authorised the next worker to enter in critical section, and loops forever.

As **mp** can arise on Power e.g. because of non-atomic writes, we know by Thm. 2 that we only need to transform one rfe pair of the cycle, and relaunch the verification. SatAbs spends 1.29 second to check it (and finds a counterexample, as previously).

PostgreSQL developers had discussed ways of fixing this, 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[3], we improved it to meet the developers' desire to maintain the current API. The final patch places 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 the mean of program transformation.

Our approach crucially relies on the definition of a generic operational model equivalent to the axiomatic one of [5]. We do not favor any style of model in particular, but we highlight the importance of having several equivalent mathematic styles to describe a field as intricate as weak memory. In addition, operational models are often the style of choice in the verification community; we contribute here to the vocabular 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

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

an existing bug (**mp**), and validated the fix proposed by the developers, including 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. Third, our work raised notable interest in both the open-source world (see our discussion with the PostgreSQL developers) and in industry (our nascent collaboration with IBM). The verification problem under weak memory is far from being solved (in particular for scalability reasons, as shown by our experiments) but we made a convincing first step.

References

1. http://research.microsoft.com/en-us/projects/poirot.
2. P. Abdulla, M. F. Atig, Y. Chen, C. Leonardsson, and A. Rezine. Counter-Example Guided Fence Insertion under TSO. In *TACAS 2012*.
3. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1995.
4. J. Alglave and L. Maranget. Stability in Weak Memory Models. In *CAV 2011*.
5. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *CAV 2010*.
6. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL 2010*.
7. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *ESOP 2012*.
8. M. F. Atig, A. Bouajjani, and G. Parlato. Getting Rid of Store-Buffers in the Analysis of Weak Memory Models. In *CAV 2011*.
9. S. Burckhardt, R. Alur, and M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI 2007*.
10. S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV 2008*.
11. L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE*, pages 331–340. ACM, 2011.
12. E. W. Dijkstra. Cooperating sequential processes. 1965.
13. A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, 2011.
14. A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV 2011*.
15. T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *FM 2006*.
16. H. Jin, T. Yavuz-Kahveci, and B. A. Sanders. Java memory model-aware model checking. In *TACAS 2012*.
17. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FM-CAD 10*.
18. M. Kuperstein, M. Vechev, and E. Yahav. Partial-Coherence Abstractions for Relaxed Memory Models. In *PLDI 2011*.
19. L. Lamport. A fast mutual exclusion algorithm.
20. L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.
21. A. Linden and P.Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN 2011*.
22. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
23. S. Owens. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP 2010*.
24. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOL 09*.
25. S. Park and D. Dill. An executable specification, analyzer and verifier for RMO. In *SPAA 95*.
26. G. L. Peterson. Myths about the mutual exclusion problem.
27. M. Rinard. Analysis of Multithreaded Programs. In *SAS 2001*.

28. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding Power Multiprocessors. In *PLDI 2011*.
29. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. In *TOPLAS 1988*.
30. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait.
31. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory model sensitive data race analysis. In *ICFEM 2004*.