Partial Orders for Efficient BMC of Concurrent Software

Jade Alglave[1], Daniel Kroening[2], and Michael Tautschnig[2,3]

[1] University College London
[2] University of Oxford
[3] Queen Mary, University of London

Abstract. The vast number of interleavings that a concurrent program can have is typically identified as the root cause of the difficulty of automatic analysis of concurrent software. Weak memory is generally believed to make this problem even harder. We address both issues by modelling programs' executions with partial orders rather than the interleaving semantics (SC). We implemented a software analysis tool based on these ideas. It scales to programs of sufficient size to achieve first-time formal verification of non-trivial concurrent systems code over a wide range of models, including SC, Intel x86 and IBM Power.

1 Introduction

Automatic analysis of concurrent programs is a practical challenge. Hardly any of the very few existing tools for concurrency will verify a thousand lines of code [21]. Most papers name the number of *thread interleavings* that a concurrent program can have as a reason for the difficulty. This view presupposes an execution model, namely *Sequential Consistency* (SC) [47], where an execution is a *total order* (more precisely an interleaving) of the instructions from different threads. The choice of SC as the execution model poses at least two problems.

First, the large number of interleavings modelling the executions of a program makes their enumeration intractable. *Context bounded* methods [59,54,45,23] (which are unsound in general) and *partial order reduction* [56,31,26] can reduce the number of interleavings to consider, but still suffer from limited scalability. Second, modern multiprocessors (e.g., Intel x86 or IBM Power) serve as a reminder that SC is an inappropriate model. Indeed, the *weak memory models* implemented by these chips allow more behaviours than SC.

We address these two issues by using *partial orders* to model executions, following [58,64,10,57]. We also aim at practical verification of concurrent programs [17,19,23]. Rarely have these two communities met. Notable exceptions are [61,62], forming with [14] the closest related work. We show that the explicit use of partial orders generalises these works to concurrency at large, from SC to weak memory, without affecting efficiency.

Our method is as follows: we map a program to a formula consisting of two parts. The first conjunct describes the data and control flow for each thread of the program; the second conjunct describes the concurrent executions of these threads as partial orders. We prove that for any satisfying assignment of this formula there is a valid execution w.r.t. our models; and conversely, any valid execution gives rise to a satisfying assignment of the formula.

Thus, given an analysis for sequential programs (the per-thread conjunct), we obtain an analysis for concurrent programs. For programs with bounded loops, we obtain a sound and complete model checking method. Otherwise, if the program has unbounded loops, we obtain an exhaustive analysis up to a given bound on loop unrollings, i.e., a bounded model checking method.

To experiment with our approach, we implement a *symbolic decision procedure* answering reachability queries over concurrent C programs w.r.t. a given memory model. We support a wide range of models, including SC, Intel x86 and IBM Power. To exercise our tool w.r.t. weak memory, we verify 4500 tests used to validate formal models against IBM Power chips [60,50]. Our tool is the first to handle the subtle *store atomicity relaxation* [4] specific to Power and ARM.

We show that mutual exclusion is not violated in a queue mechanism of the Apache HTTP server software. We confirm a bug in the worker synchronisation mechanism in PostgreSQL, and that adding two fences fixes the problem. We verify that the Read-Copy-Update mechanism of the Linux kernel preserves data consistency of the object it is protecting. For all examples we perform the analysis for a wide range of memory models, from SC to IBM Power via Intel x86.

We provide the sources of our tool, our experimental logs and our benchmarks at `http://www.cprover.org/wpo`.

2 Related Work

We start with models of concurrency, then review tools proving the absence of bugs in concurrent software, organised by techniques.

Models of concurrency Formal methods traditionally build on Lamport's SC [47]. A year earlier, Lamport defined *happens-before models* [46]. The happens-before order is the smallest partial order containing the program order and the relation between a write, and a read from this write.

These models seem well suited for analyses relative to synchronisation, e.g., [22,25,40], because the relations they define are oblivious to the implementation of the idioms. Despite happens-before being a partial order, most of [46] explains how to linearise it. Hence, this line of work often relies on a notion of total orders. Partial orders, however, have been successfully applied in verification in the context of Petri nets [53], which have been linked to software verification in [41] for programs with a small state space.

We (and [15,29,61,62]) reuse the *clocks* of [46] to build our orders. Yet we do not aim at linearisation or a transitive closure, as this leads to a polynomial overhead of redundant constraints.

Our work goes beyond the definition and simulation of memory models [32,37,63,60,50]. Implementing an executable version of the memory models is an important step, but we go further by studying the validity of systems code in C (as opposed to assembly or toy languages) w.r.t. both a given memory model and a property.

The style of the model influences the verification process. Memory models roughly fall into two classes: operational and axiomatic. The operational style models executions via interleavings, with transitions accessing buffers or queues, in addition to the memory (as on SC). Thus this approach inherits the limitations of interleaving-based verification. For example, [9] (restricted to Sun Total Store Order, TSO) bounds the number of context switches.

Other methods use operational specifications of TSO, Sun Partial Store Order (PSO) and Relaxed Memory Order (RMO) to place fences in a program [44,43,49]. Abdulla et al. [3] address this problem on an operational TSO, for finite state transition systems instead of programs. The methods of [44,43] have, in the words of [49], "severely limited scalability". The dynamic technique presented in [49] scales to 771 lines but does not aim to be sound: the tool picks an invalid execution, repairs it, then iterates.

Axiomatic specifications categorise behaviours by constraining relations on memory accesses. Several hardware vendors adopt this style [1,2] of specification; we build on the axiomatic framework of [8] (cf. Sec. 3). CheckFence [14] also uses axiomatic specifications, but does not handle the store atomicity relaxation of Power and ARM.

Running example Below we use Prog. 1 (from the TACAS Software Verification Competition [11]) as an illustration. The shared variables x and y can reach the (2N)-th Fibonacci number, depending on the interleaving of thr1 and thr2. Prog. 1 permits at least $\mathcal{O}(2^{6\mathrm{N}})$ interleavings of thr1 and thr2. In each loop iteration, thr1 reads x and then y, and then writes x; thr2 reads y and x, and then writes y. Each interleaving of these two writes yields a unique sequence of shared memory states. Swapping, e.g., the read of y in thr2 with the write of x in thr1 does not affect the memory states, but swapping the accesses to the same address does.

```
#define N 5
int x=1, y=1;

void thr1() {
  for(int k=0; k<N; ++k)
    x=x+y; }

void thr2() {
  for(int k=0; k<N; ++k)
    y=y+x; }

int main() {
  start_thread(thr1);
  start_thread(thr2);
  assert(x<=144 && y<=144);
  return 0; }
```

Prog. 1. Fibonacci from [11]

Interleaving tools Traditionally, tools are based on interleavings, and do not consider weak memory. By contrast, we handle weak memory by reasoning in terms of partial orders.

Explicit-state model checking performs a search over states of a transition system. SPIN [36], VeriSoft [30] and Java PathFinder [35,38] implement this approach; they adopt various forms of *partial order reduction* (POR) to cope with the number of interleavings.

POR reduces soundly the number of interleavings to study [56,31,26] by observing that a partial order gives rise to a class of interleavings [51], then picking only one interleaving in each class. Prog. 1 is an instance where the effect of POR is limited. We noted in Sec. 2 that amongst the $\mathcal{O}(2^{6\mathrm{N}})$ interleavings permitted by Prog. 1, only the interleavings of the writes give rise to unique sequences of states. Hence distinct interleavings of the threads representing the same interleavings of the writes are candidates for reduction. POR reduces the number of interleavings by at least $2^{2\mathrm{N}}$, but $\mathcal{O}(2^{4\mathrm{N}})$ interleavings remain.

Explicit-state methods may fail to cope with large state spaces, even in a sequential setting. Symbolic encodings [13] can help, but the state space often needs further reduction using, e.g., *bounded model checking* (BMC) [12] or *predicate abstraction* [33]. These techniques may again also be combined with POR. ESBMC [19] implements BMC. An instance of Prog. 1 has a fixed N, i.e., bounded loops. Thus BMC with N as bound is sound and complete for such an instance. ESBMC verifies Prog. 1 for $\mathrm{N} = 10$ within 30 mins (cf. Sec. 6, Fig. 8). SatAbs [17] uses predicate abstraction in a CEGAR loop; it completes no more than $\mathrm{N} = 3$ in 30 mins as it needs multiple predicates per interleaving, resulting in many refinement iterations. Our approach easily scales to, e.g., $\mathrm{N} = 50$, in less than 20 s, and more than N=300 within 30 mins, as we build only a polynomial number of constraints, at worst cubic in the number of accesses to a given shared memory address.

Non-interleaving tools Another line of tools is not based on interleavings. The existing approaches do not handle weak memory and are either incomplete (i.e., fail to prove the absence of a bug) or unsound (i.e., might miss a bug due to the assumptions they make).

Thread-modular reasoning [39,24,28,27,34] is sound, but usually incomplete. Each read presumes guarantees about the values provided by the environment. Empty guarantees amount to fully non-deterministic values, thus this is a trivially sound approach. Our translation of Sec. 4 corresponds to empty guarantees. The constraints of Sec. 5, however, make our encoding complete.

In Prog. 1, if we guarantee `x<=144 && y<=144`, the problem becomes trivial, but finding this guarantee automatically is challenging. Threader [34] fails for N=1 (cf. Sec. 6, Fig. 8).

Context bounded methods fix an arbitrary bound on *context switches* [59,54,45,23]. This supposes that most bugs happen with few context switches. Our method does not make this restriction. Moreover, we believe that there is no obvious generalisation of these works to weak memory, other than instrumentation as [9] does for TSO, i.e., adding information to a program so that its SC executions simulate its weak ones. We used our tool in SC mode, and applied the instrumentation of [9] to it. On average, the instrumentation is 9 times more costly (cf. Sec. 6, Fig. 8).

In Prog. 1, we need at least N context switches to disprove the assertion `assert(x<=143 && y<=143)` (or any upper bound to x and y that is the (2N)-th Fibonacci number minus 1). The hypothesis of the approach (i.e., small context bounds suffice to find a bug) does not apply here; Poirot fails for $\mathrm{N} \geq 1$ (cf Sec. 6, Fig. 8).

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`
Allowed? `r1=0; r2=0`	

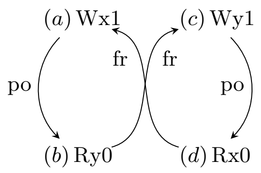

Fig. 1. Store Buffering (**sb**)

Our work relates the most to [14,29,61,62]; we discuss [29] below and detail [14,61,62] in Sec. 5.7. These works use axiomatic specifications of SC to compose the distinct threads. CheckFence [14] models SC with total orders and transitive closure constraints; [61,62] use partial orders like us. [61,62] note redundancies of their constraints, but do not explain them; our semantic foundations (Sec. 3) allow us both to explain their redundancies and avoid them (cf. Sec. 5.7).

The encodings of [14,61,62] are $\mathcal{O}(\mathrm{N}^3)$ for N shared memory accesses *to any address*; [29] is quadratic, but in the number of threads times the number of per-thread transitions, which may include arbitrary many local accesses. Our encoding is $\mathcal{O}(\mathrm{M}^3)$, with M the maximal number of events for a *single address*. By contrast, the encodings of [29,61,62] quantify over all addresses. Prog. 1 has two addresses only, but the difference is already significant: $(6\mathrm{N})^3$ for [14,29,61,62] vs. $2 \times (3\mathrm{M})^3$ in our case, i.e. 1/4 of the constraints (cf. Sec. 6, Fig. 7 for other case studies).

3 Context: Axiomatic Memory Model

We use the framework of [8], which provably embraces several *architectures*: SC [47], Sun TSO (i.e. the x86 model [55]), PSO and RMO, Alpha, and a fragment of Power. We present this framework via *litmus tests*, as shown in Fig. 1.

The keyword *allowed* asks if the architecture permits the outcome "`r1=1;r2=0; r3=1;r4=0`". This relates to the event graphs of this program, composed of relations over *read and write memory events*. A store instruction (e.g. `x` $\leftarrow$ `1` on P_0) corresponds to a write event ((a) Wx1), and a load (e.g. `r1` $\leftarrow$ `y` on P_0) to a read ((b) Ry0). The validity of an execution boils down to the absence of certain cycles in the event graph. Indeed, an architecture allows an execution when it represents a *consensus* amongst the processors. A cycle in an event graph is a potential violation of this consensus.

If a graph has a cycle, we check if the architecture *relaxes* some relations. The consensus ignores relaxed relations, hence becomes acyclic, i.e. the architecture allows the final state. In Fig. 1, on SC where nothing 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 for (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,

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`		

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

$$\begin{tikzcd}
(a)\,\mathrm{Rx1} \arrow[d, bend right, "\mathrm{po}"'] & (c)\,\mathrm{Ry1} \arrow[d, bend right, "\mathrm{po}"'] & (e)\,\mathrm{Wx1} \arrow[ll, bend right, "\mathrm{rf}"'] & (f)\,\mathrm{Wy1} \arrow[ll, bend right, "\mathrm{rf}"'] \\
(b)\,\mathrm{Ry0} \arrow[rrru, bend right, "\mathrm{fr}"'] & (d)\,\mathrm{Rx0} \arrow[ru, bend right, "\mathrm{fr}"'] & &
\end{tikzcd}$$

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

and a value. We represent each instruction by the events it issues. In Fig. 2, we associate the store $x \leftarrow \texttt{1}$ on processor P_2 to the event (e) Wx1.

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 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`, 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 [4] (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 [4]), 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 are relaxed (e.g. write-read pairs on x86), i.e. only a subset of po is guaranteed to occur in this order.

When a relation is not 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 consider the write serialisation ws and the from-read relation fr to be always 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 are relaxed) but nothing else. We denote the safe subset of read-from, i.e. the read-from relation globally agreed on by all processors, by grf.

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

From the validity of executions we deduce a comparison of architectures: We say that an architecture A_2 is *stronger* than another one A_1 when the executions valid on A_2 are valid on A_1. Equivalently we would say that A_1 is *weaker* than A_2. Thus, SC is stronger than any other architecture discussed above.

4 Symbolic event structures

For an architecture A and *one* execution witness X, the framework of Sec. 3 determines if X is valid on A. To prove reachability of a program state, we need to reason about all its executions. To do so efficiently, we use symbolic representations capturing all possible executions in a single constraint system. We then apply SAT or SMT solvers to decide if a valid execution exists for A, and, if so, get a satisfying assignment corresponding to an execution witness.

As said in Sec. 1, we build two conjuncts. The first one, ssa, represents the data and control flow per thread. The second, pord, captures the communications between threads (cf. Sec. 5). We include a reachability property in ssa; the program has a valid execution violating the property iff $\mathsf{ssa} \wedge \mathsf{pord}$ is satisfiable.

We mostly use *static single assignment form* (SSA) of the input program to build ssa (cf. [42] for details). In this SSA variant, each equation is augmented with a *guard*: the guard is the disjunction over all conjunctions of branching guards on paths to the assignment. To deal with concurrency, we use a fresh index for each occurrence of a given shared memory variable, resulting in a fresh symbol in the formula. CheckFence [14] and [61,62] use a similarly modified encoding.

Together with ssa, we build a *symbolic event structure* (ses). As detailed below, it captures basic program information needed to build the second conjunct pord in Sec. 5. Fig. 3 illustrates this section: the formula ssa on top corresponds to the ses beneath.

main	P_0	P_1	P_2	P_3
$x_0 = 0$				
$\wedge\, y_0 = 0$	$\wedge\, r1^1_0 = x_1$	$\wedge\, r3^2_0 = y_2$	$\wedge\, x_3 = 1$	$\wedge\, y_3 = 1$
	$\wedge\, r2^1_0 = y_1$	$\wedge\, r4^2_0 = x_2$		
$\wedge$ prop				

(i_0) Wxx_0				
(i_1) Wyy_0	(a) Rxx_1	(c) Ryy_2	(e) Wxx_3	(f) Wyy_3
	(b) Ryy_1	(d) Rxx_2		

Fig. 3. The formula ssa for **iriw** (Fig. 2) with prop $= (r1^1_0 = 1 \wedge r2^1_0 = 0 \wedge r3^2_0 = 1 \wedge r4^2_0 = 0)$, and its ses (guards omitted since all true)

Static single assignment form (SSA) To encode ssa we use a variant of SSA [20] and loop unrolling. The details of this encoding are in [42], except for differences in the handling of shared memory variables, as explained below.

In SSA, each occurrence of a program variable is annotated with an index. We turn assignments in SSA form into equalities, with distinct indexes yielding distinct symbols in the resulting equation. For example, the assignment `x:=x+1` results in the equality $x_1 = x_0 + 1$. We use unique indexes for assignments in loops via loop unrolling: repeating `x:=x+1` twice yields $x_1 = x_0 + 1$ and $x_2 = x_1 + 1$. Control flow join points yield additional equations involving the guards of branches merging at this point (see [42] for details).

In concurrent programs, we also need to consider join points due to communication between threads, i.e., *concurrent SSA form* (CSSA) [48]. To deal with weaker models, we use a fresh index for each occurrence of a given shared memory variable, resulting in a fresh symbol in the formula. Thus, each occurrence may take non-deterministic values, i.e. this approach over-approximates the behaviours of a program. If `x` is shared in the above example, the modified SSA encoding of the second loop unrolling becomes $x_3 = x_2 + 1$, breaking any causality between the first loop iteration (encoded as $x_1 = x_0 + 1$) and the second one. Sinha and Wang [61,62] use the same approach, but since they consider SC only, their use of fresh indexes may produce more symbols than necessary.

By adding the negation of the reachability property to be checked to our (over-approximating) SSA equations, we obtain a formula ssa that is satisfiable if there exists a (concurrent) counterexample violating the property. As this is an over-approximation, the converse need not be true, i.e., a satisfying assignment of ssa may constitute a spurious counterexample. Sec. 5 restores precision using the pord constraints derived from the ses.

In ssa, memory addresses map to unique symbols via the (symbolic) pointer dereferencing of [42, Sec. 4]. In the weak memory case, we ensure this by using analyses sound for this setting [6].

The top of Fig. 3 gives ssa for Fig. 2. We print a column per thread, vertically following the control flow, but it forms a single conjunction. Each occurrence of a program variable carries its SSA index as a subscript. Each occurrence of the shared memory variables x and y has a unique SSA index. Here we omit the guards, as this program does not use branching or loops.

From SSA to symbolic event structures A symbolic event structure (ses) $\gamma \triangleq (\mathbb{S}, \mathsf{po})$ is a set $\mathbb{S}$ of *symbolic events* and a *symbolic program order* po. A symbolic event holds a *symbolic value* instead of a concrete one as in Sec. 3. We define $\mathrm{g}(e)$ to be the Boolean guard of a symbolic event e, which corresponds to the guard of the SSA equation as introduced above. We use these guards to build the executions of Sec. 3: a guard evaluates to true if the branch is taken, false otherwise. The symbolic program order $\mathsf{po}(\gamma)$ gives a list of symbolic events per thread of the program. The order of two events in $\mathsf{po}(\gamma)$ gives the program order in a concrete execution if both guards are true.

Note that $\mathsf{po}(\gamma)$ is an implementation-dependent linearisation of the branching structure of a thread, induced by the path merging applied while constructing the SSA form. For instance, `if` e_1 `then` e_2 `else` e_3 could be linearised as either (e_1, e_2, e_3) or (e_1, e_3, e_2) as any two events of a concrete execution (e_1 and e_2, or e_1 and e_3) remain in program order. The original branching structure, i.e., the unlinearised symbolic program order induced by the control flow graph, is maintained in the relation $\mathsf{po\text{-}br}(\gamma)$. For the above example, $\mathsf{po\text{-}br}(\gamma)$ contains (e_1, e_2) and (e_1, e_3).

We build the ses γ alongside the SSA form, as follows. Each occurrence of a shared program variable on the right-hand side of an assignment becomes a *symbolic read*, with the SSA-indexed variable as symbolic value, and the guard is taken from the SSA equation. Similarly, each occurrence of a shared program variable on the left-hand side becomes a *symbolic write*. Fences do not affect memory states in a sequential setting, hence do not appear in SSA equations. We simply add a fence event to the ses when we see a fence. We take the order of assignments per thread as program order, and mark thread spawn points.

At the bottom of Fig. 3, we give the ses of **iriw**. Each column represents the symbolic program order, per thread. We use the same notation as for the events of Sec. 3, but values are SSA symbols. Guards are omitted again, as they all are trivially true. We depict the thread spawn events by starting the program order in the appropriate row. Note that we choose to put the two initialisation writes in program order on the main thread.

From symbolic to concrete event structures To relate to the models of Sec. 3, we *concretise* symbolic events. A satisfying assignment to $\mathsf{ssa} \wedge \mathsf{pord}$, as computed by a SAT or SMT solver, induces, for each symbolic event, a concrete value (if it is a read or a write) and a valuation of its guard (for both accesses and fences). A valuation V of the symbols of ssa includes the values of each symbolic event. Since guards are formulas that are part of ssa, V allows us to evaluate the guards as well. For a valuation V, we write $\mathrm{conc}(e_s, \mathsf{V})$ for the concrete event corresponding to e_s, if there is one, i.e., if $\mathrm{g}(e_s)$ evaluates to true under V.

The concretisation of a set $\mathbb{S}$ of symbolic events is a set $\mathbb{E}$ of concrete events, as in Sec. 3, s.t. for each $e \in \mathbb{E}$ there is a symbolic version e_s in $\mathbb{S}$. We write $\mathrm{conc}(\mathbb{S}, \mathsf{V})$ for this concrete set $\mathbb{E}$. The concretisation $\mathrm{conc}(\mathsf{r}_s, \mathsf{V})$ of a symbolic relation r_s is the relation $\{(x, y) \mid \exists (x_s, y_s) \in \mathsf{r}_s . x = \mathrm{conc}(x_s, \mathsf{V}) \wedge y = \mathrm{conc}(y_s, \mathsf{V})\}$.

Given an ses γ, $\mathrm{conc}(\gamma, \mathsf{V})$ is the event structure (cf. Sec. 3), whose set of events is the concretisation of the events of γ w.r.t. V, and whose program order is the concretisation of $\mathsf{po}(\gamma)$ w.r.t. V. For example, the graph of Fig. 2 (erasing the rf and fr relations) is a concretisation of the ses of **iriw** (cf. Fig. 3).

5 Encoding the communication and weak memory relations symbolically

For an architecture A and an ses γ, we need to represent the communications (i.e., rf, ws and fr) and the weak memory relations (i.e., $\mathsf{ppo}_A, \mathsf{grf}_A$ and ab_A) of Sec. 3. We encode them as a formula pord, s.t. $\mathsf{ssa} \wedge \mathsf{pord}$ is satisfiable iff there is an execution valid on A violating the property encoded in ssa. We avoid transitive closures to obtain a small number of constraints. We start with an informal overview of our approach, then describe how we encode partial orders, and finally detail the encoding for each relation of Sec. 3.

Overview We present our approach on **iriw** (Fig. 2) and its ses γ (Fig. 3). In Fig. 2, we represent only one possible execution, namely the one corresponding to the (non-SC) final state of the test at the top of the figure. In this section, we generate constraints representing all the executions of **iriw** on a given architecture. We give these constraints, for the address x in Fig. 4 in the SC case (for brevity we skip y, analogous to x). Weakening the architecture removes some constraints: for Power, we omit the (rf-grf) and (ppo) constraints. For TSO, all constraints are the same as for SC.

In Fig. 4, each symbol c_{ab} is a *clock constraint*, representing an ordering between the events a and b. A variable s_{wr} represents a read-from between the write w and the read r.

The constraints of Fig. 4 represent the preserved program order (cf. Sec. 5.4), e.g., on SC or TSO the read-read pairs (a, b) on P_0 (ppo P_0) and (c, d) on P_1 (ppo P_1), but nothing on Power. We generate constraints for the read-from (cf. Sec. 5.1), for example (rf-some x); the first conjunct $s_{i_0 a} \vee s_{ea}$ concerns the read a on P_0. This means that a can read either from the initial write i_0 or from the write e on P_2. The selected read-from pair also implies equalities of the

(rf-val x)	$(s_{i_0a} \Rightarrow x_1 = x_0) \wedge (s_{i_0d} \Rightarrow x_2 = x_0) \wedge$ $(s_{ea} \Rightarrow x_1 = x_3) \wedge (s_{ed} \Rightarrow x_2 = x_3)$
(rf-grf x)	$(s_{i_0a} \Rightarrow c_{i_0a}) \wedge (s_{ea} \Rightarrow c_{ea}) \wedge$ $(s_{i_0d} \Rightarrow c_{i_0d}) \wedge (s_{ed} \Rightarrow c_{ed})$
(rf-some x)	$(s_{i_0a} \vee s_{ea}) \wedge (s_{i_0d} \vee s_{ed})$
(ws x)	$\neg c_{i_0e} \Rightarrow c_{ei_0}$
(fr x)	$((s_{i_0a} \wedge c_{i_0e}) \Rightarrow c_{ae}) \wedge ((s_{i_0d} \wedge c_{i_0e}) \Rightarrow c_{de}) \wedge$ $((s_{ea} \wedge c_{ei_0}) \Rightarrow c_{ai_0}) \wedge ((s_{ed} \wedge c_{ei_0}) \Rightarrow c_{di_0})$
(ppo main)	$c_{i_0i_1}$ (ppo P_0) c_{ab} (ppo P_1) c_{cd}

Fig. 4. Partial order constraints for address x in Fig. 2 on SC

values written and read (rf-val x): for instance, s_{i_0a} implies that x_1 equals the initialisation x_0. The architecture-independent constraints for write serialisation (cf. Sec. 5.2) and from-read (cf. Sec. 5.3) are specified as (ws x) and (fr x); (ws y) and (fr y) are analogous. As there are no fences in **iriw**, we do not generate any memory fence constraints (cf. Sec. 5.5).

We represent the execution of Fig. 2 as follows. For (e, a) and $(i_0, d) \in \mathsf{grf}$, we have the constraint $s_{ea} \Rightarrow c_{ea}$ and $s_{i_0d} \Rightarrow c_{i_0d}$ in (rf-grf x). This means that a reads from e (as witnessed by s_{ea}), and that we record that e is ordered before a in grf (as witnessed by c_{ea}); *idem* for d and i_0. To represent $(d, e) \in \mathsf{fr}$, we pick the appropriate constraint in (fr x), namely $(s_{i_0d} \wedge c_{i_0e}) \Rightarrow c_{de}$. This reads "if d reads from i_0 and i_0 is ordered before e (in ws, because i_0 and e are two writes to x), then d is ordered before e (in fr)."

Together with (ppo P_0) and (ppo P_1), these constraints represent the execution in Fig. 2. We cannot find a satisfying assignment of these constraints, as this leads to both a before b (by (ppo P_0)) and b before a (by (fr y), (rf-grf y), (ppo P_1), (fr x) and (grf x)). On Power, however, we neither have the ppo nor the grf constraints, hence we can find a satisfying assignment.

Symbolic partial orders We associate each symbolic event x of an ses γ with a unique *clock* variable clock_x (cf. [46,61]) ranging over the naturals. For two events x and y, we define the Boolean *clock constraint* as $c_{xy} \triangleq (\mathrm{g}(x) \wedge \mathrm{g}(y)) \Rightarrow \mathsf{clock}_x < \mathsf{clock}_y$ ("$<$" being less-than over the integers). We encode a relation r over the symbolic events of γ as the formula $\phi(\mathsf{r})$ defined as the conjunction of the clock constraints c_{xy} for all $(x, y) \in \mathsf{r}$, i.e., $\phi(\mathsf{r}) \triangleq \bigwedge_{(x,y)\in\mathsf{r}} c_{xy}$.

Let C be a valuation of the clocks of the events of γ. Let V be a valuation of the symbols of the formula ssa associated to γ. As noted in Sec. 4, V gives us concrete values for the events of γ, and allows us to evaluate their guards. We show below that (C, V) satisfies $\phi(\mathsf{r})$ iff the concretisation of r w.r.t. V is acyclic, provided that this relation has *finite prefixes*.

A prefix of x in a relation r is a (possibly infinite) list $S = [x_0, x_1, x_2, \ldots]$ s.t. $x = x_0$ and for all i, $(x_{i+1}, x_i) \in \mathsf{r}$ (observe that the prefix is reversed w.r.t. the order imposed by the relation). The relation r has finite prefixes if for each x, there is a bound $l \in \mathbb{N}$ to the cardinality of the prefixes of x in r.

We write $\mathrm{card}(S)$ for the cardinality of a list $S = [x_0, x_1, x_2, \dots]$, i.e., $\mathrm{card}(S) \triangleq \mathrm{card}(\{x \mid \exists i.x = x_i\})$. We write $\mathrm{pref}(\mathsf{r}, x)$ for the set of prefixes of x in r. Formally, r has finite prefixes when $\forall x.\exists l.\forall S \in \mathrm{pref}(\mathsf{r}, x).\, \mathrm{card}(S) < l$. In our proofs and in Alg. 4 we denote the concatenation of two lists S_1 and S_2 by $S_1{+}{+}S_2$.

In the following, we allow symbolic relations with infinite prefixes provided their concretisations have finite prefixes. Thus we do not consider executions with an infinite past, or running for more steps than the cardinality of $\mathbb{N}$. Our first lemma justifies why checking the acyclicity of a concrete relation amounts to checking the satisfiability of the formula encoding this relation symbolically:

Lemma 1. *(C, V) satisfies $\phi(\mathsf{r})$ iff $\mathrm{conc}(\mathsf{r}, \mathsf{V})$ is acyclic and has finite prefixes.*

Proof. *$\Rightarrow$: We let $\mathsf{r_c} = \mathrm{conc}(\mathsf{r}, \mathsf{V})$. One can show by induction that $(*)$ if (C, V) satisfies $\phi(\mathsf{r})$ then for all $(x, y) \in \mathsf{r_c}^+$, c_{xy} is true. Now, suppose $\phi(\mathsf{r})$ satisfied, and as a contradiction, $\mathsf{r_c}$ cyclic, i.e., $\exists x.(x, x) \in \mathsf{r_c}^+$. Thus c_{xx} is true by $(*)$; this contradicts the irreflexivity of $<$ over the integers.*

Now we show that $\mathsf{r_c}$ has finite prefixes, i.e., for each x we give a bound l over all $S \in \mathrm{pref}(\mathsf{r_c}, x)$. As a contradiction take $S = [x_0, \dots x_n] \in \mathrm{pref}(\mathsf{r_c}, x)$ s.t. $x = x_0$ and $\mathrm{card}(S) > \mathsf{clock}_x$. Thus for all i, we have $(x_{i+1}, x_i) \in \mathsf{r_c}$ and $\mathsf{clock}_{x_{i+1}} < \mathsf{clock}_{x_i}$ by $(*)$. Since $n \geq \mathrm{card}(S)$, $\mathrm{card}(S) > \mathsf{clock}_x$ and $\mathsf{clock}_{x_0} = \mathsf{clock}_x$, we have $\mathsf{clock}_{x_n} < 0$, which contradicts the fact that our clocks are naturals. Thus for each x we can take $l = \mathsf{clock}_x$.

$\Leftarrow$: Let $\mathsf{r_c} = \mathrm{conc}(\mathsf{r}, \mathsf{V})$. For all e s.t. $\mathrm{g}(e) = false$, take $\mathsf{clock}_e = 0$. Thus c_{xy} is true if $\mathrm{g}(x)$ or $\mathrm{g}(y)$ is false. Now, have $(x, y) \in \mathsf{r}$ with both guards true, i.e., $(x, y) \in \mathsf{r_c}$. Take clock_x to be the maximal cardinality of the S in $\mathrm{pref}(\mathsf{r_c}, x)$, idem *for y. We want to prove $\mathsf{clock}_x < \mathsf{clock}_y$. Take S s.t. $\mathsf{clock}_x = \mathrm{card}(S)$. From $(x, y) \in \mathsf{r_c}$, we have $[y]{+}{+}S \in \mathrm{pref}(\mathsf{r_c}, y)$. Now, $\mathrm{card}([y]{+}{+}S) \leq \mathsf{clock}_y$ by maximality of clock_y. It suffices to prove $\mathrm{card}(S) < \mathrm{card}([y]{+}{+}S)$. Suppose $\mathrm{card}(S) \geq \mathrm{card}([y]{+}{+}S)$. Then y appears in S. Thus $(y, x) \in \mathsf{r_c}^+$ since S is a prefix of x; as $(x, y) \in \mathsf{r_c}$ by hypothesis, we have a cycle in $\mathsf{r_c}$.*

The formula $\phi(\mathsf{r}_1 \cup \mathsf{r}_2)$ is equivalent to $\phi(\mathsf{r}_1) \wedge \phi(\mathsf{r}_2)$. Thus we encode unions of relations, e.g., $\mathsf{ghb}_A \triangleq \mathsf{ws} \cup \mathsf{fr} \cup \mathsf{grf}_A \cup \mathsf{ppo}_A \cup \mathsf{ab}_A$, as the conjunction of their respective encodings. By Lem. 1, the acyclicity of ghb_A corresponds to the satisfiability of $\phi(\mathsf{ghb}_s)$, where ghb_s is a symbolic encoding of ghb_A. To form $\phi(\mathsf{ghb}_s)$, we form the conjunction of the formulas $\phi(\mathsf{r})$, for r being a symbolic encoding of ws, fr, grf_A, ppo_A and ab_A.

We now present these encodings, in that order. Sec. 3 also relies on the program order per location for the uniproc check, and the dependencies for the thin check, omitted for brevity. We compute them alongside the preserved program order; they use independent sets of clock variables, but the same clock constraints.

We define auxiliaries over symbolic events: $\mathrm{tid}(e)$ is the thread identifier of e, $\mathrm{addr}(e)$ the memory address read from or written to (e.g., x for (e) Rxy), and $\mathrm{val}(e)$ its (symbolic) value. Each algorithm outputs constraints, whose conjunction we add to pord.

```
   input: $\gamma$, $A$    output: $C_{\mathrm{wf}}, C_{\mathsf{rf}}, C_{\mathsf{grf}}$
1  $\mathsf{reads} := \{(\alpha, \{r_1 \dots r_n\}) \mid r_i$ is read $\wedge\ \mathrm{addr}(r_i) = \alpha\}$
2  $\mathsf{writes} := \{(\alpha, \{w_1 \dots w_n\}) \mid w_i$ is write $\wedge\ \mathrm{addr}(w_i) = \alpha\}$
3  $C_{\mathsf{rf}} := \emptyset$; $C_{\mathsf{grf}} := \emptyset$; $C_{\mathrm{wf}} := \emptyset$
4  foreach $\alpha$ s.t. $\exists R, W.(\alpha, R) \in \mathit{reads} \wedge (\alpha, W) \in \mathit{writes}$ do
5      foreach $r \in R$ do
6          $\mathsf{rf\_some} := \emptyset$
7          foreach $w \in W$ do
8              if $(r, w) \notin \mathit{po}(\gamma)$ then
9                  $\mathsf{rf\_some} := \mathsf{rf\_some} \cup \{s_{wr}\}$
10                 $C_{\mathrm{wf}} := C_{\mathrm{wf}} \cup \{s_{wr} \Rightarrow (\mathrm{g}(w) \wedge \mathsf{val}(r) = \mathsf{val}(w))\}$
11                 $C_{\mathsf{rf}} := C_{\mathsf{rf}} \cup \{s_{wr} \Rightarrow c_{wr}\}$
12                 if $(w, r)$ not relaxed on $A$ and $\mathrm{tid}(w) \neq \mathrm{tid}(r)$ then
13                     $C_{\mathsf{grf}} := C_{\mathsf{grf}} \cup \{s_{wr} \Rightarrow c_{wr}\}$
14         $C_{\mathrm{wf}} := C_{\mathrm{wf}} \cup \{\mathrm{g}(r) \Rightarrow \bigvee_{s \in \mathsf{rf\_some}} s\}$
```

Algorithm 1: Constraints for read-from

For each algorithm we state and prove a lemma about its correctness. These follow the scheme of Lem. 1, i.e. we show the encoding correct for any satisfying valuation of clocks and ssa. Thus we will introduce symbolic encodings of sets $\mathsf{r}(\gamma)$, where membership in the set is given by a formula and thus depends on the actual valuation under C and V.

5.1 Read-from

For an architecture A and an ses γ, Alg. 1 encodes the read-from (resp. safe read-from) as the set of constraints C_{rf} (resp. C_{grf}). Following Sec. 3, we add constraints to C_{grf} depending on: first, the relation being within one thread or between distinct threads (derivable from $\mathrm{tid}(w)$ and $\mathrm{tid}(r)$); second, whether A exhibits store buffering, store atomicity relaxation, or both.

Alg. 1 groups the reads and writes by address, in the sets reads and writes (lines 1 and 2). For **iriw**, $\mathsf{reads} = \{(\mathrm{x}, \{a, d\}), (\mathrm{y}, \{b, c\})\}$ and $\mathsf{writes} = \{(\mathrm{x}, \{i_0, e\}), (\mathrm{y}, \{i_1, f\})\}$.

The next step forms the potential read-from pairs. To that end, Alg. 1 introduces a free Boolean variable s_{wr} for each pair (w, r) of write and read to the same address (line 9), unless such a pair contradicts program order (line 8). Indeed, if (w, r) is in rf and (r, w) is in po, this violates the uniproc check of Sec. 3.

The variable $\mathsf{rf\_some}$, initialised in line 6, collects the variables s_{wr} in line 9. For **iriw**, the memory address x, and the read a, we have $\mathsf{rf\_some} = \{s_{i_0 a}, s_{ea}\}$, i.e., the read a can read either from i_0 (the initial write to x), or from the write e on P_2.

Following Sec. 3, each read must read from some write. We ensure this at line 14, by gathering in C_{wf}, for a given r, the union of all the potential read-from s_{wr} collected in $\mathsf{rf\_some}$.

Going back to **iriw**, recall from Sec. 4 that an event has a guard indicating the branch of the program it comes from. In **iriw**, the guard of a is true (as all the others), i.e., the read a is concretely executed. Hence there exists a write (either i_0 or e) from which a reads, as expressed by the constraint $s_{i_0a} \vee s_{ea}$ formed at line 14.

If s_{wr} evaluates to true (i.e., r reads from w), we record the *value constraint* $\text{val}(r) = \text{val}(w)$ in the set C_{wf} (line 10). For **iriw**, we obtain the following for x: $(s_{i_0a} \Rightarrow x_1 = x_0) \wedge\ (s_{i_0d} \Rightarrow x_2 = x_0) \wedge\ (s_{ea} \Rightarrow x_1 = x_3) \wedge\ (s_{ed} \Rightarrow x_2 = x_3)$. The constraint $s_{i_0a} \Rightarrow x_1 = x_0$ reads "if s_{i_0a} is true (i.e., a reads from i_0) then the value x_1 read by a equals the value x_0 written by i_0."

The constraint added to C_{rf} is such that only if s_{wr} evaluates to true, the clock constraint c_{wr} is enforced (line 11). For **iriw** we add the following to C_{rf}, for the address x: $(s_{i_0a} \Rightarrow c_{i_0a}) \wedge\ (s_{ea} \Rightarrow c_{ea}) \wedge (s_{i_0d} \Rightarrow c_{i_0d}) \wedge\ (s_{ed} \Rightarrow c_{ed})$.

If (w, r) is not relaxed on A, we also add its clock constraint c_{wr} to C_{grf} (line 13). In **iriw**, all reads read from an external thread. Thus on an architecture that does not relax store atomicity (i.e., stronger than Power), we add the constraints that we added to C_{rf} to C_{grf} as well. On Power, C_{grf} remains empty.

We now write grf_A for both the function over concrete relations given by the definition of A as in Sec. 3, and the corresponding function over symbolic relations. Given an architecture A, we have $\mathsf{grf}_A(\mathsf{r}) = \{(w, r) \in \mathsf{r} \mid (w, r) \text{ is not relaxed on } A\}$. For example if A is TSO, all thread-local read-from pairs are relaxed: $\mathsf{grf}_A(\mathsf{r}) = \{(w, r) \in \mathsf{r} \mid \text{tid}(w) \neq \text{tid}(r)\}$. We write $(w, r) \in \text{WR}_\alpha$ when w writes to an address α and r reads from the same α, and $\mathsf{prf}(\gamma) \triangleq \{(w, r) \in \bigcup_\alpha \text{WR}_\alpha \mid (r, w) \notin \mathsf{po}(\gamma)\}$. We write $\mathsf{rf}(\gamma)$ for the set $\{(w, r) \in \mathsf{prf}(\gamma) \mid s_{wr}\}$ (with s_{wr} of Alg. 1), and $\mathsf{grf}(\gamma)$ for $\mathsf{grf}_A(\mathsf{rf}(\gamma))$. Note that we build the external safe read-from ($\mathsf{grfe}(\gamma)$) only, i.e., between two events from distinct threads. We compute the internal one as part of ppo_A, in Alg. 4.

Given an ses γ, Alg. 1 outputs $C_{\mathsf{rf}}, C_{\mathsf{grf}}$ and C_{wf}. Let WR be a valuation of the s_{wr} variables of γ. We write $\mathsf{inst}(\mathsf{rf}(\gamma), \mathsf{WR})$ (resp. $\mathsf{inst}(\mathsf{grf}(\gamma), \mathsf{WR})$) for $\mathsf{rf}(\gamma)$ (resp. $\mathsf{grfe}(\gamma)$) where WR instantiates the s_{wr} variables (thus $\mathsf{rf}(\gamma)$ is a symbolic encoding of the set as noted before this sub-section; we use this notation similarly in the remainder of this section). We show that Alg. 1 gives the clock constraints encoding grf (we omit the corresponding lemma for rf):

Lemma 2. (C, V, WR) *satisfies* $\bigwedge_{c \in C_{\text{wf}} \cup C_{\mathsf{grf}}} c$ *iff* (C, V) *satisfies*
i) for all r *s.t.* $\text{g}(r)$ *is true, there is* w *s.t.* $(w, r) \in \mathsf{inst}(\mathsf{rf}(\gamma), WR)$ *and*
ii) for all $(w, r) \in \mathsf{inst}(\mathsf{rf}(\gamma), WR)$*,* $\text{g}(w)$ *is true and* $\text{val}(w) = \text{val}(r)$ *and*
iii) $\bigwedge_{(w,r) \in \mathsf{inst}(\mathsf{grfe}(\gamma), WR)} c_{wr}$.

Proof. An induction on R, W *s.t.* $(\alpha, R) \in \mathsf{reads}$ *and* $(\alpha, W) \in \mathsf{writes}$ *for some address* α*, then union for all* α *shows that* $C_{\mathsf{grf}} = \{s_{wr} \Rightarrow c_{wr} \mid (w, r) \in \mathsf{prf}(\gamma) \cap \mathsf{grfe}_A\}$*, and* $C_{\text{wf}} = \bigcup_{r \text{ is read}} \{\text{g}(r) \Rightarrow \bigvee_{(w,r) \in \mathsf{prf}(\gamma)} s_{wr}\} \cup \{s_{wr} \Rightarrow (\text{g}(w) \wedge \text{val}(r) = \text{val}(w)) \mid (w, r) \in \mathsf{prf}(\gamma)\}$*. The first component of* C_{wf} *is equivalent to i); the second to ii).* C_{grf} *is equivalent to iii).*

The model described in Sec. 3 suggests that rf must be encoded to be exclusive, i.e., to link a read to only one write. An explicit encoding thereof, however,

```
input: $\gamma$   output: $C_{ws}$
1 $\mathsf{writes} := \{(\alpha, \{w_1 \dots w_n\}) \mid w_i$ is write $\wedge\ \mathrm{addr}(w_i) = \alpha\}$
2 $C_{\mathsf{ws}} := \emptyset$; foreach $\alpha$ s.t. $\exists W.(\alpha, W) \in$ writes do
3   foreach $w \in W$ do
4     foreach $w' \in W s.t.\ \mathrm{tid}(w') \neq \mathrm{tid}(w)$ do
5       $C_{\mathsf{ws}} := C_{\mathsf{ws}} \cup \{\neg c_{ww'} \Rightarrow c_{w'w}\}$
```

Algorithm 2: Constraints for write serialisation

would be redundant, as this is already enforced by ws and fr. Hence it suffices to consider *at least one* write per read, as Alg. 1 does:

Lemma 3. $\mathrm{uniproc}(E, X) \Rightarrow \forall r.\neg(\exists w \neq w'.(w, r) \in \mathsf{rf} \wedge (w', r) \in \mathsf{rf})$

Proof. By contradiction, have $w \neq w'$ s.t. $(w, r) \in \mathsf{rf}$ and $(w', r) \in \mathsf{rf}$. By totality of ws*, $(w, w') \in \mathsf{ws}$ or $(w', w) \in \mathsf{ws}$. W.l.o.g. have $(w, w') \in \mathsf{ws}$. Then $(r, w') \in \mathsf{fr}$, i.e., a cycle in $\mathsf{rf} \cup \mathsf{fr}$: w', r, w', forbidden by* uniproc.

5.2 Write serialisation

Given an ses γ, Alg. 2 encodes the write serialisation ws as the set of constraints C_{ws}. By definition, ws is a total order over writes to a given address. Alg. 2 implements the totality by ensuring that for two writes $w \neq w'$ to the same address either $c_{ww'}$ or $c_{w'w}$ holds. For implementation reasons we choose to express this as $\neg c_{ww'} \Rightarrow c_{w'w}$ rather than $c_{ww'} \vee c_{w'w}$.

Alg. 2 groups the writes per address. For each address α and write w to α (lines 2 and 3) we choose another write w' to α (line 4), and build the disjunction of clock constraints over w and w' (line 5). For **iriw** we have $\mathsf{writes} = \{(\mathrm{x}, \{i_0, e\}), (\mathrm{y}, \{i_1, f\})\}$, and the constraints: $(\neg c_{i_0 e} \Rightarrow c_{e i_0}) \wedge (\neg c_{i_1 f} \Rightarrow c_{f i_1})$.

Note that we build the external ws only (wse). With WW_α the pairs of writes to the address α, and $\mathsf{ws}(\gamma)$ the set $\{(w, w') \in \bigcup_\alpha WW_\alpha \mid c_{w'w} = \mathrm{false}\}$, we have $\mathsf{wse}(\gamma) \triangleq \mathsf{ws}(\gamma) \cap \{(w, w') \mid \mathrm{tid}(w) \neq \mathrm{tid}(w')\}$. We compute the thread-local ws as part of ppo_A, in Alg. 4. Given an input γ of Alg. 2, we now characterise the clock constraints given by C_{ws}. Basically we show that Alg. 2 gives the clock constraints enconding ws. The proof (omitted for brevity) is by induction as for Lem. 2:

Lemma 4. (C, V) *satisfies* $\bigwedge_{c \in C_{ws}} c$ *iff it satisfies* $\bigwedge_{(w, w') \in \mathsf{wse}} c_{ww'}$.

We quantify over all pairs of writes to the same address to build ws. Thus for w_0, w_1, w_2 in ws in a concrete execution, we build $(w_0, w_1), (w_1, w_2)$ and the redundant (w_0, w_2) in the symbolic world. This is inherent to the totality of ws.

5.3 From-read

Given an ses γ, Alg. 3 encodes from-read as the set of constraints C_{fr}. Recall that $(r, w) \in \mathsf{fr}$ means $\exists w'.(w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}$. The existential quantifier

```
input: $\gamma$    output: $C_{\mathsf{fr}}$
1 reads := $\{(\alpha,\{r_1\ldots r_n\}) \mid r_i$ is read $\wedge$ addr$(r_i)=\alpha\}$
2 writes := $\{(\alpha,\{w_1\ldots w_n\}) \mid w_i$ is write $\wedge$ addr$(w_i)=\alpha\}$
3 $C_{\mathsf{fr}} := \emptyset$
4 foreach $\alpha$ s.t. $\exists R, W.(\alpha, W) \in$ writes, $(\alpha, R) \in$ reads do
5     foreach $(w, w') \in W \times W s.t. w' \neq w$ do
6         foreach $r \in R$ with tid$(r) \neq$ tid$(w)$ do
7             $C_{\mathsf{fr}} := C_{\mathsf{fr}} \cup \{(s_{w'r} \wedge c_{w'w} \wedge \mathrm{g}(w)) \Rightarrow c_{rw}\}$
```

Algorithm 3: Constraints for from-read

corresponds to a disjunction: $\bigvee_{w' \text{is write}} (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}$. Since this disjunction can be large, which is undesirable in the expression simplification used in the implementation, we rewrite it as a conjunction of small implications, each of which are simplified in isolation: $\bigwedge_{w' \text{is write}} ((r, w) \in \mathsf{fr} \Leftarrow (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws})$. Thus Alg. 3 encodes from-read as a conjunction of the premise variables s_{wr} of C_{rf} and clock variables $c_{ww'}$ of C_{ws} introduced in Alg. 1 and 2.

Again, we collect the sets of reads and writes per address. Alg. 3 considers triples (w', w, r) of events to the same address, where (w', w) is in the write serialisation, and (w', r) is in read-from. We enumerate the pairs of writes in line 5, and then pick a read in line 6. For each such triple we add in line 7 the clock constraint c_{rw} under the premise that i) $(w', r) \in \mathsf{rf}$, witnessed by $s_{w'r}$, ii) $(w', w) \in \mathsf{ws}$, witnessed by $c_{w'w}$, and that iii) the write w actually takes place in a concrete execution, i.e., $\mathrm{g}(w)$ evaluates to true.

For **iriw** all guards are true. For x, we obtain: $(s_{i_0a} \wedge c_{i_0e}) \Rightarrow c_{ae}) \wedge\ ((s_{i_0d} \wedge c_{i_0e}) \Rightarrow c_{de}) \wedge\ ((s_{ea} \wedge c_{ei_0}) \Rightarrow c_{ai_0}) \wedge\ ((s_{ed} \wedge c_{ei_0}) \Rightarrow c_{di_0}$. For example, $(s_{i_0a} \wedge c_{i_0e}) \Rightarrow c_{ae}$, reads "if s_{i_0a} is true (i.e., if a reads from i_0), and if c_{i_0e} is true (i.e., $(i_0, e) \in \mathsf{ws}$) then c_{ae} is true (i.e., a is in fr before e)."

Given an ses γ, Alg. 3 outputs C_{fr}. Note that we compute here the external from-read only (fre), and the internal one as part of ppo_A, in Alg. 4. We show that Alg. 3 gives the clock constraints encoding fr. The (omitted) proof is as for Lem. 2:

Lemma 5. *(C, V, WR) satisfies $\bigwedge_{c \in C_{fr}} c$ iff (C, V) satisfies $\bigwedge_{(r,w) \in inst(fre(\gamma), WR)} c_{rw}$.*

The fr defined above, together with ws, does introduce possible redundancies: given $(w_0, r) \in \mathsf{rf}$ with $(w_0, w_1) \in \mathsf{ws}$ and $(w_1, w_2) \in \mathsf{ws}$, we have both $(r, w_1) \in \mathsf{fr}$ and $(r, w_2) \in \mathsf{fr}$ – but the latter is redundant as the same ordering is implied by $(r, w_1) \in \mathsf{fr}$ and $(w_1, w_2) \in \mathsf{ws}$. We could thus, instead, build a fragment of fr, which we write fr_0. We define fr_0 as $\{(r, w_1) \mid \exists w_0.(w_0, r) \in \mathsf{rf} \wedge (w_0, w_1) \in \mathsf{ws} \wedge \nexists w'.((w_0, w') \in \mathsf{ws} \wedge (w', w_1) \in \mathsf{ws})\}$. In Fig. 5, (r, w_1) is in fr_0 but not (r, w_2), because there is a write (w_1) in ws between the write w_0 from which r reads and w_2. One can show that $(r, w) \in \mathsf{fr}$ if $(r, w) \in \mathsf{fr}_0$ or

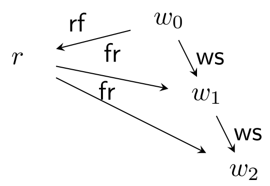

Fig. 5. fr derives from rf and ws

```
input: $\gamma$, $A$    output: $C_{ppo}$
1  $C_{\mathsf{ppo}} := \emptyset$; foreach $S \in po(\gamma) \wedge S \neq \emptyset$ do
2     $S = [e]{+}{+}S' \cap \{e \mid e \text{ is not fence}\}$
3     $\mathsf{chains} := [(e, \emptyset)]$; $R := \text{true}$
4     foreach $e' \in S'$ do
5        $T' := \emptyset$
6        foreach $(e'', T'') \in$ chains s.t. there is no $r$ s.t.
                 $(e'', r) \in T'$ and $((\mathrm{g}(e') \wedge \mathrm{g}(e'') \wedge R) \Rightarrow r)$ do
7           $r_{e''e'} := \mathsf{not\_relax}\, A\, \gamma\, (e'', e')$
8           if $r_{e''e'}$ is satisfiable then
9              $C_{\mathsf{ppo}} := C_{\mathsf{ppo}} \cup \{r_{e''e'} \Rightarrow c_{e''e'}\}$
10             $T' := T' \cup \{(e'', r_{e''e'})\}$
11             foreach $(e, r) \in T''$ do
12                if $\exists r'.(e, r') \in T'$ then
13                   $R := R \wedge (\rho \Leftrightarrow r' \vee (r_{e''e'} \wedge r))$
14                   $T' := \{(e, \rho)\} \cup T' \setminus (e, r')$
15                else $T' := \{(e, r_{e''e'} \wedge r)\} \cup T'$
16       $\mathsf{chains} := [(e', T')]{+}{+}[\mathsf{chains}]$
```

Algorithm 4: Constraints for preserved program order

there exists w' s.t. $(r, w') \in \mathsf{fr}_0$ and $(w', w) \in \mathsf{ws}$, i.e., we can generate fr from fr_0 and ws.

5.4 Preserved program order

For an architecture A and an ses γ, Alg. 4 encodes the preserved program order as the set C_{ppo}. In Sec. 3, the function ppo_A, which is part of the definition of A, determines if A relaxes a pair (e, e') in program order in a concrete execution. For example, RMO and Power relax read-read pairs, but PSO and stronger do not.

We reuse the notation ppo_A for the function collecting non-relaxed pairs in symbolic program order. Unlike in Sec. 3, the non-relaxed pairs in symbolic program order also include the internal safe read-from, internal write serialisation, internal from-read, and the orderings due to Power's `isync` fence. We generate these constraints here, rather than in Alg. 1–3, to limit the redundancies. We write $\mathsf{ppo}_A(\gamma)$ for $\mathsf{ppo}_A(\mathsf{po\text{-}br}(\gamma))$, or only ppo_A if γ is clear from the context.

Alg. 4 avoids building redundant transitive closure constraints, taking into account the guards of events: for two events e_1, e_2, we build a constraint iff $(e_1, e_2) \in \mathsf{ppo}_A(\gamma)$. If, e.g., $\mathsf{ppo}_A(\mathsf{po\text{-}br}) = \mathsf{po\text{-}br}$ (on SC), Alg. 4 creates constraints only for neighbouring events in $\mathsf{po\text{-}br}(\gamma)$ in each control flow branch of the program.

As SSA and loop unrolling yield $\mathsf{po}(\gamma)$ (i.e., lists of symbolic events per thread) rather than $\mathsf{po\text{-}br}(\gamma)$ (the corresponding DAG), we cannot construct C_{ppo} by analysing control flow branches of the program. Building C_{ppo} from $\mathsf{po}(\gamma)$ requires some more work.

To build ppo_A, Alg. 4 uses the variable chains, a list of pairs (y, T). For a given y, its companion set T contains the events x occurring before y in ${\mathsf{ppo}_A}^+$ together with a formula r that characterises all paths of ${\mathsf{ppo}_A}^+$ between x and y. We build r from formulas $r_{e''e'}$ asserting that $(e'', e') \in \mathsf{ppo}_A$, describing individual steps (e'', e') of a path between x and y.

We compute the formula $r_{e''e'}$ at line 7, using the function $\mathsf{not\_relax}$. Given an ses γ and a pair (e'', e'), $\mathsf{not\_relax}\, A\ \gamma\ (e'', e')$ returns a formula $r_{e''e'}$ expressing the condition under which (e'', e') is not relaxed. For PSO or stronger models, $\mathsf{not\_relax}$ only needs to take the direction of the events and their addresses into account. For instance, TSO relaxes write-read pairs, but nothing else. If a pair is necessarily relaxed, $\mathsf{not\_relax}$ returns false, otherwise $\mathsf{not\_relax}\, A\ \gamma\ (e'', e') = \mathrm{g}(e'') \wedge \mathrm{g}(e')$. For models weaker than PSO, such as Alpha, RMO or Power, $\mathsf{not\_relax}$ has to determine data- and control dependencies, and handle Power's `isync` fence. We resolve data dependencies via a definition-use data flow analysis [5] on the program part in program order between the two events. Control dependencies use the data dependency analysis to test whether there exists a branching instruction in program order between the events such that the branching decision is in data dependency with the first event. For `isync`, the approach is similar, except that in addition there must be an `isync` in program order between the branch and the second event. We then add the guard of the fence to the conjunction returned by $\mathsf{not\_relax}$.

For a given e', we initialise its companion set T' at line 5, then increment it in lines 10–15. In line 14, we use fresh variables ρ constrained in the formula R (line 13) to avoid repeating sub-formulas, as is standard in, e.g., CNF encodings [16]. In line 7 we compute the condition $r_{e''e'}$ for (e'', e') not being relaxed on A for each e'' in chains (unless skipped for transitivity, see below). We generate the constraint $r_{e''e'} \Rightarrow c_{e''e'}$ iff $r_{e''e'}$ is satisfiable (line 9), i.e., (e'', e') is not relaxed on A.

Now, suppose e_1, e_2, e_3 on the same thread all in ppo_A; the companion set of e_2 is $\{(e_1, r_{e_1e_2})\}$, because $(e_1, e_2) \in \mathsf{ppo}_A$ and there is no other event before e_1 on the thread. Suppose that Alg. 4 has already built the beginning of the chain formed by e_1, e_2 and e_3, so that $\mathsf{chains} = [(e_2, \{(e_1, r_{e_1e_2})\}), (e_1, \emptyset)]$ (observe that the chains are in reverse order of po). At line 4, for each remaining e' on a given thread, i.e., e_3 in our example, Alg. 4 follows lines 5–9 and adds a constraint w.r.t. the immediate predecessor e_2 of e_3 in ppo_A. The subsequent elements of chains (e_1 in our example) are also candidates for a clock constraint.

We do not add any constraint if (e_1, e_3) is guaranteed to be in ${\mathsf{ppo}_A}^+$, as follows. Any remaining element of chains that belongs to the companion set T'' of e'' is added to T' at lines 11–15. As an instance, recall that e_1 is in the companion set of e_2. Thus, after generating the constraint $c_{e_2e_3}$ at line 9, we add e_1 with its transitivity condition $r_{e_2e_3} \wedge r_{e_1e_2}$ to T' at line 15. Then, line 6 iterates over the rest of chains, i.e., $(e_1, \emptyset)$. With the updated set T' the test $(e_1, r_{e_1}) \in T'$ yields $r_{e_1} = r_{e_2e_3} \wedge r_{e_1e_2}$, and thus amounts to checking the validity of $(\mathrm{g}(e_3) \wedge \mathrm{g}(e_1)) \Rightarrow (r_{e_2e_3} \wedge r_{e_1e_2})$. Remember that, unless there is an `isync`, the conditions r_{xy} amount to conjunctions over guards, hence in our example we are checking the validity of $(\mathrm{g}(e_3) \wedge \mathrm{g}(e_1)) \Rightarrow \mathrm{g}(e_3) \wedge \mathrm{g}(e_2) \wedge \mathrm{g}(e_1)$. If all three events e_1, e_2 and e_3 are on the same control flow branch, the implication is valid because all guards are equal. This makes the test of line 6 fail and (e_1, e_3) will not be considered for adding another constraint $c_{e_1 e_3}$, which would have been redundant. When the implication is not valid, the test of line 6 succeeds; then we add another constraint $c_{e_1 e_3}$, as this is not redundant here.

This elaboration on guards is essential as witnessed by the following variant of our example: assume, in contrast to the above, that e_2 is not a dominator of e_3 on the control flow graph. This might occur in a program fragment `(if` e_1 `then` e_2`);` e_3, where the guard of e_2 would be different from that of e_1 or e_3. If we were to skip (e_1, e_3) as above, the constraints would be insufficient to enforce the order of e_1 before e_3 when $\mathrm{g}(e_2)$ evaluates to false. In this case, the premises $\mathrm{g}(e_1) \wedge \mathrm{g}(e_2)$ and $\mathrm{g}(e_2) \wedge \mathrm{g}(e_3)$ of $c_{e_1 e_2}$ and $c_{e_2 e_3}$, respectively, are false, hence the clock constraints $\mathsf{clock}_{e_1} < \mathsf{clock}_{e_2}$ and $\mathsf{clock}_{e_2} < \mathsf{clock}_{e_3}$ are not enforced, leaving the order of (e_1, e_3) unconstrained.

We illustrate Alg. 4 on the ses γ of **iriw** (cf. Fig. 3). Alg. 4 proceeds over $\mathsf{po}(\gamma)$, equal to $\{[i_0, i_1], [a, b], [c, d], [e], [f]\}$ for **iriw**. Given a non-empty list S of $\mathsf{po}(\gamma)$, e.g., $S = [a, b]$ corresponding to P_0, the first non-fence event a of S initialises at line 3 the variable chains (explained below in detail). The loop at line 4 proceeds with the tail S' of the list S. Thus for P_0 at this point we have $\mathsf{chains} = [(a, \emptyset)]$ and Alg. 4 proceeds with $S' = [b]$.

The contents of chains depend on the architecture A, as **iriw** shows. For P_0, recall that $\mathsf{chains} = [(a, \emptyset)]$ and only b remains in S'. If A relaxes read-read pairs, e.g., RMO or weaker, then (a, b) is relaxed. Thus we do not add any clock constraint to C_{ppo} at line 9 and eventually $\mathsf{chains} = [(b, \emptyset), (a, \emptyset)]$ in line 16. If A does not relax read-read pairs, e.g., PSO or stronger, we add c_{ab} to C_{ppo} at line 9 and add a with the guard conjunction true to T' at line 15. Thus $\mathsf{chains} = [(b, \{(a, \mathrm{true})\}), (a, \emptyset)]$. Let us now characterise the output of Alg. 4, given an input ses γ:

Lemma 6. *Alg. 4 outputs* $\{r_{xy} \Rightarrow c_{xy} \mid (x, y) \in \mathsf{ppo}_A\}$.

Proof. We write L_1 (resp. L_2) for the loop from line 4 to 16 (resp. 6 to 15). L_1 maintains the invariant that $S = \mathrm{rd}(\mathsf{chains}) {+}{+} S'$, where rd reverses its argument and deletes T for each element (e, T) of its argument. We write $\mathrm{path}^{e}_{x,y}(e_1, \ldots, e_n)$ when there is a path from x to y in $\mathsf{ppo}_A(\gamma)$ passing by e, i.e., $e_1 = x$ and $e_n = y$ and $\forall i.(e_i, e_{i+1}) \in \mathsf{ppo}_A(\gamma)$ and $\exists i. e_i = e$. L_2 maintains the invariant that $T' = \bigcup_{e \in [e'', e']} T^{e'}_{e}$, where $e \in [e'', e']$ means $(e'', e) \in \mathsf{po} \wedge (e, e') \in \mathsf{po}$, and $T^{e'}_{e} = \{(x, r_x) \mid r_x = \bigvee_{\mathrm{path}^{e}_{x,y}(e_1, \ldots, e_n)} \bigwedge_{1 \le i \le n} r_{e_i e_{i+1}}\}$. We conclude by double inclusion of C_{ppo} and $\{r_{xy} \Rightarrow c_{xy} \mid (x, y) \in \mathsf{ppo}_A\}$, omitted for brevity.

Since the r_{xy} are guard conditions, we just need to evaluate the guards to evaluate them. We show that Alg. 4 gives the clock constraints encoding ppo; the proof is immediate by Lem. 6:

Lemma 7. (C, V) *satisfies* $\bigwedge_{c \in C_{\mathsf{ppo}}} c$ *iff it satisfies* $\bigwedge_{(x,y) \in \mathsf{ppo}_A} c_{xy}$.

```
input: $\gamma$, $A$    output: $C_{\mathsf{ab}'}$
1  $C_{\mathsf{ab}'} := \emptyset$; foreach $S \in po(\gamma) \wedge S \neq \emptyset$ do
2    fences $:= \{s \mid s \in S \wedge \text{s is fence}\}$
3    foreach $e \in S \setminus fences$ do
4      foreach $s \in fences$ do
5        if $(e, s) \in po(\gamma)$ then
6          $C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{\text{g}(s) \Rightarrow c_{es}\}$
7          if $A$ is not store atomic then
8            foreach $(w, e)$ being a w-r pair s.t.  $\text{addr}(w) = \text{addr}(e)$ and
               $\text{tid}(w) \neq \text{tid}(e)$ do
9              $C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{(\text{g}(s) \wedge s_{we}) \Rightarrow c_{ws}\}$
10       else $C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{\text{g}(s) \Rightarrow c_{se}\}$
11         if $A$ is not store atomic then
12           foreach $(e, r)$ being a w-r pair s.t.  $\text{addr}(e) = \text{addr}(r)$ and
               $\text{tid}(e) \neq \text{tid}(r)$ do
13             $C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{(\text{g}(s) \wedge s_{er}) \Rightarrow c_{sr}\}$
```

Algorithm 5: Constraints for memory fences

5.5 Memory fences and cumulativity

Given an architecture A and an ses γ, Alg. 5 encodes the fence orderings as the set $C_{\mathsf{ab}'}$. A fence s potentially induces orderings over all (e, e') s.t. e is in po before s and e' after s, which is quadratic in the number of events in po for each fence. Cumulativity constraints depend on the read-from to appear in the concrete event structure, and again these are paired with all events before or after (in po) a fence. We alleviate this with the fence events (see below). The implementation supports x86's `mfence` and Power's `sync`, `lwsync` and `isync`. We handle `isync` as part of ppo in Alg. 4. We first present x86's `mfence` and Power's `sync`, then `lwsync`.

Fences `mfence` and `sync` Alg. 5 applies its procedure to $\mathsf{po}(\gamma)$ (line 1). For example, assume `sync` fences between the read-read pairs of P_0 and P_1 of **iriw**, associated with the fences events s_0 and s_1. We then have $\mathsf{po}(\gamma) = \{[i_0, i_1], [a, s_0, b], [c, s_1, d], [e], [f]\}$.

For each list S of $\mathsf{po}(\gamma)$ (i.e., per thread), we compute at line 2 the set fences, containing the fence events of S. For **iriw**, fences is empty for P_2 and P_3. For P_0, we have fences $= \{s_0\}$, and $\{s_1\}$ for P_1. We test at line 5 for each pair (e, s) s.t. e is a non-fence event and s is fence whether (e, s) is in program order, or rather (s, e). We then build the according non-cumulative constraints, and constraints for A-cumulativity (for (e, s) in program order) or B-cumulativity (otherwise).

For non-cumulativity, if e is before (resp. after) s in program order, Alg. 5 produces at line 6 the clock constraint c_{es} (resp. c_{se} at line 10). In **iriw**, all guards are true, hence we generate c_{as_0} (resp. c_{cs_1}) for the event a (resp. c) in po before the fence s_0 (resp. s_1) on P_0 (resp. P_1). Line 10 generates $c_{s_0 b}$ (resp. $c_{s_1 d}$) for b (resp. d), in po after the fence s_0 (resp. s_1) on P_0 (resp. P_1).

If A relaxes store atomicity, we build cumulativity constraints. For A-cumulativity, Alg. 5 adds at line 9 the constraint $s_{we} \Rightarrow c_{ws}$, for each (w, e) s.t. e is in po before the fence s, and e reads from the write w. The constraint reads "if $\mathrm{g}(s)$ is true (i.e., the fence is concretely executed) and if s_{we} is true (i.e., e reads from w), then c_{ws} is true (i.e., there is a global ordering, due to the fence s, from w to s)". All other constraints, i.e., the actual ordering of w before some event e' in po *after* s, follow by transitivity. We handle B-cumulativity in a similar way, given in lines 12 and 13.

As Power relaxes store atomicity, the `sync` fences between the read-read pairs of **iriw** create A-cumulativity constraints, namely for s_0 (and analogous ones for s_1): $(s_{i_0 a} \Rightarrow c_{i_0 s_0}) \wedge (s_{ea} \Rightarrow c_{es_0})$.

If we were not using fence events, we would create a clock constraint $c_{we'}$ for every e' in program order after the fence s to implement Sec. 3, for each fence s. Thus the non-cumulative part would be cubic already, whereas fence events yield a quadratic number at most. For cumulativity, we would obtain a constraint for every pair (r, e') s.t. $(w, r) \in \mathsf{rfe}$ and r is in po before the fence s. The resulting number of constraints is the number of such pairs (r, e') times the number of pairs (w, r), i.e., cubic in the number of events *per fence* s. Furthermore cases of both A- and B-cumulativity at the same fence s need to be taken into account, resulting in even higher complexity. Fence events, however, reduce all these cases, including the combined one, to cubic complexity (all triples of external writes, reads, and fence events).

Fence `lwsync` As `lwsync` does not order write-read pairs (cf. Sec. 3), we need to avoid creating a constraint c_{wr} between a write w and a read r separated by an `lwsync`. To do so, we use two distinct clock variables clock_s^r and clock_s^w for an `lwsync` s. This avoids the wrong transitive constraint c_{wr} implied by c_{ws} and c_{sr}. Fig. 6 illustrates this setup: the write-read pair (w_1, r_2) will not be ordered by any of the constraints, but all other pairs are ordered.

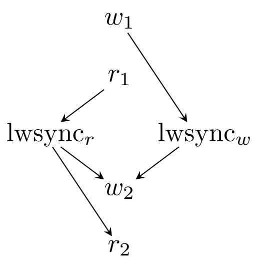
$$\begin{tikzcd}
 & & w_1 \arrow[ddr] & \\
 & r_1 \arrow[dl] & & \\
\mathrm{lwsync}_r \arrow[dr] \arrow[ddr] & & & \mathrm{lwsync}_w \arrow[dll] \\
 & w_2 & & \\
 & r_2 & &
\end{tikzcd}$$

Fig. 6. Constraints for `lwsync`

To create a clock constraint in lines 6, 9, 10, or 13, we then pick one or both of the clock variables, as follows. If e is a read, the clock constraint is $\mathsf{clock}_e < \mathsf{clock}_s^r$ when e is before s, i.e., lines 6 or 9 (or $\mathsf{clock}_s^r < \mathsf{clock}_e$ if e is after, i.e., lines 10 or 13). If e is a write preceding s (i.e., lines 6 or 9), the clock constraint is $\mathsf{clock}_e < \mathsf{clock}_s^w$. Finally, if e is a write after s, i.e., lines 10 or 13, the clock constraint is the conjunction $(\mathsf{clock}_s^w < \mathsf{clock}_e) \wedge (\mathsf{clock}_s^r < \mathsf{clock}_e)$. To make `lwsync` non-cumulative (cf. footnote in Sec. 3), we just need to disable the lines 8,9,12 and 13.

In **iriw**, if we use `lwsync` instead of `sync` as discussed above, we obtain the following constraints: $(\mathsf{clock}_a < \mathsf{clock}_{s_0}^r) \wedge (\mathsf{clock}_{s_0}^r < \mathsf{clock}_b) \wedge (s_{i_0 a} \Rightarrow \mathsf{clock}_{i_0} < \mathsf{clock}_{s_0}^w) \wedge (s_{ea} \Rightarrow \mathsf{clock}_e < \mathsf{clock}_{s_0}^w)$. These constraints will *not* order the writes i_0 or e with the read b, because i_0 and e are ordered w.r.t. to $\mathsf{clock}_{s_0}^w$, but b is only ordered w.r.t. the distinct $\mathsf{clock}_{s_0}^r$. This corresponds to the fact that placing `lwsync` fences in **iriw** does not forbid the non-SC execution.

Given an ses γ, Alg. 5 outputs $C_{\mathsf{ab}'}$. We let $\mathsf{rfe}(\gamma)$ be $\{(w, r) \in \bigcup_\alpha \mathrm{WR}_\alpha \mid \mathrm{tid}(w) \neq \mathrm{tid}(r) \wedge s_{wr}\}$. We write $\mathsf{ab}'(\gamma)$ for $\{(e_1, e_2) \mid \mathrm{nc}'(e_1, e_2) \vee \mathrm{ac}'(e_1, e_2) \vee$

$\mathrm{bc}'(e_1, e_2)\}$, where $\mathrm{nc}'(e_1, e_2)$ corresponds to non-cumulativity, i.e., $(e_1, e_2) \in \mathsf{po}(\gamma) \wedge ((\mathrm{g}(e_1) \wedge e_1 \text{ is fence}) \vee (\mathrm{g}(e_2) \wedge e_2 \text{ is fence})) \wedge$not both e_1 and e_2 are fences, $\mathrm{ac}'(e_1, e_2)$ to A-cumulativity, i.e., $\exists r.(e_1, r) \in \mathsf{rfe}(\gamma) \wedge (r, e_2) \in \mathsf{po}(\gamma) \wedge \mathrm{g}(e_2) \wedge$ e_2 is fence, and $\mathrm{bc}'(e_1, e_2)$ corresponds to B-cumulativity, i.e., $\exists w.(w, e_2) \in \mathsf{rfe}(\gamma) \wedge$ $(e_1, w) \in \mathsf{po}(\gamma) \wedge \mathrm{g}(e_1) \wedge e_1$ is fence. We show that Alg. 5 gives the clock constraints encoding ab'. The proof is immediate like for Lem. 2:

Lemma 8. *(C, V, WR) satisfies $\bigwedge_{c \in C_{\mathsf{ab}'}} c$ iff (C, V) satisfies $\bigwedge_{(x,y) \in \mathit{inst}(\mathsf{ab}'(\gamma), WR)} c_{xy}$.*

We let $\mathsf{ab}(\gamma)$ be the symbolic version of ab in Sec. 3, i.e., we let $\mathrm{nc}(e_1, s, e_2)$ be $\mathrm{g}(s) \wedge s$ is fence $\wedge\, (e_1, s) \in \mathsf{po}(\gamma) \wedge (s, e_2) \in \mathsf{po}(\gamma)$, $\mathrm{ac}(e_1, s, e_2)$ be $\exists r.(e_1, r) \in \mathsf{rfe}(\gamma) \wedge \mathrm{nc}(r, s, e_2)$ and $\mathrm{bc}(e_1, s, e_2)$ be $\exists w.\, \mathrm{nc}(e_1, s, w) \wedge (w, e_2) \in \mathsf{rfe}(\gamma)$. We only prove this encoding sound w.r.t. Sec. 3, as ab' is more fine-grained than ab (to see why, note that one cannot express $\mathrm{nc}'(e_1, e_2)$ as a combination of nc, ac or bc). Yet we prove our overall encoding complete in Thm. 1.

Lemma 9. *If (C, V, WR) satisfies $\bigwedge_{c \in C_{\mathsf{ab}'}} c$ then (C, V) satisfies $\bigwedge_{(e_1,e_2) \in \mathit{inst}(\mathit{ab}(\gamma), WR)} c_{e_1 e_2}$.*

Proof. *We give only the case $\mathsf{lwsync}(\gamma)$. Take $(e_1, e_2) \in \mathsf{lwsync}(\gamma)$, i.e., there is an `lwsync` s s.t. $\mathrm{nc}(e_1, s, e_2)$ or $\mathrm{ac}(e_1, s, e_2)$ or $\mathrm{bc}(e_1, s, e_2)$. In the nc case, we know that s is a fence and $\mathrm{g}(s)$ is true, and $(e_1, s) \in \mathit{po}(\gamma)$ and $(s, e_2) \in \mathit{po}(\gamma)$, i.e., $\mathrm{nc}'(e_1, s)$ and $\mathrm{nc}'(s, e_2)$. Thus $c_{e_1 s}$ and $c_{s e_2}$ are in $C_{\mathsf{ab}'}$. Now, by definition of $\mathsf{lwsync}(\gamma)$, $(e_1, e_2) \notin \mathrm{WR}$. For $(e_1, e_2) \in \mathrm{WW}$, $c_{e_1 s}$ is $\mathsf{clock}_{e_1} < \mathsf{clock}^w_s$ and $c_{s e_2}$ is $\mathsf{clock}^w_s < \mathsf{clock}_{e_2}$. Thus $\mathsf{clock}_{e_1} < \mathsf{clock}_{e_2}$, i.e., $c_{e_1 e_2}$ holds. Writing RR for the read-read pairs, take $(e_1, e_2) \in \mathrm{RR}$. Thus $c_{e_1 s}$ is $\mathsf{clock}_{e_1} < \mathsf{clock}^r_s$ and $c_{s e_2}$ is $\mathsf{clock}^r_s < \mathsf{clock}_{e_2}$. Hence $\mathsf{clock}_{e_1} < \mathsf{clock}_{e_2}$, i.e., $c_{e_1 e_2}$ holds. For $(e_1, e_2) \in \mathrm{RW}$, $c_{e_1 s}$ is $\mathsf{clock}_{e_1} < \mathsf{clock}^r_s$ and $c_{s e_2}$ is $(\mathsf{clock}^w_s < \mathsf{clock}_{e_2}) \wedge (\mathsf{clock}^r_s < \mathsf{clock}_{e_2})$. Thus $\mathsf{clock}_{e_1} < \mathsf{clock}_{e_2}$, i.e., $c_{e_1 e_2}$ holds. In the ac case, s is a fence and $\mathrm{g}(s)$ is true, and there is r s.t. $(e_1, r) \in \mathit{rfe}(\gamma)$ and $\mathrm{nc}(r, s, e_2)$. Thus e_1 is a write (source of a rf), and since $(e_1, e_2) \notin \mathrm{WR}$ (by definition of $\mathsf{lwsync}(\gamma)$), e_2 is a write. So $\mathrm{ac}'(e_1, s)$, and $\mathrm{nc}'(s, e_2)$, i.e., $c_{e_1 s}$ and $c_{s e_2}$ hold. We are back to the WW case. In the bc case, s is a fence and $\mathrm{g}(s)$ is true, and there is w s.t. $\mathrm{nc}(e_1, s, w)$ and $(w, e_2) \in \mathit{rfe}(\gamma)$. Thus e_2 is a read (target of a rf), and since $(e_1, e_2) \notin \mathrm{WR}$, e_1 is a read. So $\mathrm{nc}'(e_1, s)$ and $\mathrm{bc}'(s, e_2)$, i.e., $c_{e_1 s}$ and $c_{s e_2}$ hold. We are back to the RR case.*

5.6 Soundness and completeness of the encoding

Given an architecture A and a program, the procedure of Sec. 4 and Sec. 5 outputs a formula $\mathsf{ssa} \wedge \mathsf{pord}$ and an ses γ. This formula provably encodes the executions of this program valid on A and violating the property encoded in ssa in a sound and complete way. Proving this requires proving that any assignment to the system corresponds to a valid execution of the program, and vice versa. This result requires three steps, one for uniproc, one for thin and one for the acyclicity of ghb. By lack of space, we show only the last one. Given an ses γ, we write ϕ for $\bigwedge_{c \in C_{\mathsf{ppo}} \cup C_{\mathsf{grf}} \cup C_{\mathrm{wf}} \cup C_{\mathsf{ws}} \cup C_{\mathsf{ab}'}} c$:

Theorem 1. *The formula* $\mathsf{ssa} \wedge \phi$ *is satisfiable iff there are* V*, a valuation to the symbols of* ssa*, and a well formed* X *s.t.* $\mathsf{ghb}_A(\mathrm{conc}(\gamma, \mathsf{V}), X)$ *is acyclic and has finite prefixes.*

Proof. *Let* $(\mathsf{C}, \mathsf{V}, \mathsf{WR})$ *be a satisfying assignment of* $\mathsf{ssa} \wedge \phi$*. By Lem. 7, 2, 4, 5 and 8, we know that* $(\mathsf{C}, \mathsf{V}, \mathsf{WR})$ *satisfies* ϕ *iff i) for all* r *s.t.* $\mathrm{g}(r)$ *is true, there is* w *s.t.* $(w, r) \in \mathsf{inst}(\mathsf{rf}(\gamma), \mathsf{WR})$ *and ii) for all* $(w, r) \in \mathsf{inst}(\mathsf{rf}(\gamma), \mathsf{WR})$*,* $\mathrm{g}(w)$ *is true and* $\mathrm{val}(w) = \mathrm{val}(r)$ *and iii)* $(\mathsf{C}, \mathsf{V}, \mathsf{WR})$ *satisfies* $\phi(\mathsf{ppo}_A(\gamma)) \wedge \phi(\mathsf{inst}(\mathsf{grf}(\gamma), \mathsf{WR})) \wedge \phi(\mathsf{wse}(\gamma)) \wedge \phi(\mathsf{inst}(\mathsf{fr}(\gamma), \mathsf{WR})) \wedge \phi(\mathsf{inst}(\mathsf{ab}'(\gamma), \mathsf{WR}))$*.*

$\Rightarrow$*: Take* $X = (\mathrm{conc}(\mathsf{inst}(\mathsf{rf}(\gamma), \mathsf{WR}), \mathsf{V})), (\mathrm{conc}(\mathsf{ws}(\gamma), \mathsf{V}))$*. Note that i) and ii), together with Lem. 3, imply that* $\mathsf{rf}(X)$ *is well formed. For* $\mathsf{ws}(X)$*, this comes from the totality of* $\mathsf{ws}(\gamma)$ *over writes to the same address, implied by the shape of* C_{ws} *(cf. Alg. 2) for the external* ws*, and by the totality of* $\mathsf{po}(\gamma)$ *for the internal* ws*.*

By Lem. 8 and 9, iii) says that $(\mathsf{C}, \mathsf{V}, \mathsf{WR})$ *satisfies* $\phi(r)$*, with* $\mathsf{r} = \mathsf{ppo}_A(\gamma) \cup \mathsf{inst}(\mathsf{grf}_A(\gamma), \mathsf{WR}) \cup \mathsf{ws}(\gamma) \cup \mathsf{inst}(\mathsf{fr}(\gamma), \mathsf{WR}) \cup \mathsf{inst}(\mathsf{ab}(\gamma), \mathsf{WR})$*. By Lem. 1, since* $\mathsf{ghb}_A(\mathrm{conc}(\gamma, \mathsf{V}), X)$ *is* $\mathrm{conc}(\mathsf{r}, \mathsf{V})$*, we have our result.*

$\Leftarrow$*: We let* E *be* $\mathrm{conc}(\gamma, \mathsf{V})$*. Take* WR *s.t.* s_{wr} *is true iff* $(w, r) \in \mathsf{rf}(X)$*.* $\mathsf{rf}(X)$ *being well-formed implies i) and ii).*

We let $\mathsf{ghb}'_A(E, X)$ *be* $\mathsf{r}_1 \cup \mathsf{ab}'(E, X)$*, with* $\mathsf{r}_1 = \mathsf{ppo}_A(E) \cup \mathsf{grf}_A(X) \cup \mathsf{ws}(X) \cup \mathsf{fr}(E, X)$*. Note that* $\mathsf{ghb}_A(E, X)$ *is* $\mathsf{r}_1 \cup \mathsf{ab}(E, X)$*. We show below that the acyclicity of* $\mathsf{ghb}_A(E, X)$ *implies the acyclicity of* $\mathsf{ghb}'_A(E, X)$ *(*idem *for finite prefixes). Then we take* $\mathsf{r} = \mathsf{ghb}'_A(E, X)$ *in Lem. 1, and take* C *as in Lem. 1. Hence we have* (C, V) *satisfying* $\phi(\mathsf{ppo}_A(\gamma)) \wedge \phi(\mathsf{inst}(\mathsf{grf}(\gamma), \mathsf{WR})) \wedge \phi(\mathsf{wse}(\gamma)) \wedge \phi(\mathsf{inst}(\mathsf{fr}(\gamma), \mathsf{WR})) \wedge \phi(\mathsf{inst}(\mathsf{ab}'(\gamma), \mathsf{WR}))$*, namely iii); our result follows.*

We let $\mathsf{r}_2 = \{(e_1, e_2) \in \mathsf{ab}'; \mathsf{ab}' \mid$ *neither* e_1 *nor* e_2 *is a fence*$\})$*. We write* $(x, y) \in \mathsf{r}; \mathsf{r}'$ *for* $\exists z.(x, z) \in \mathsf{r} \wedge (z, y) \in \mathsf{r}'$*. One can show:* $(*)$ *if* $(e_1, e_2) \in \mathsf{ghb}'^{+}$*, we have* (e_1, e_2) *in* $\mathsf{ab}'; (\mathsf{r}_1{}^{+} \cup \mathsf{r}_2{}^{+})^{+}$*, or* $(\mathsf{r}_1{}^{+} \cup \mathsf{r}_2{}^{+})^{+}; \mathsf{ab}'$*, or* $(\mathsf{r}_1{}^{+} \cup \mathsf{r}_2{}^{+})^{+}$*.*

Acyclicity: by contradiction, take a cycle in $\mathsf{ghb}'_A(E, X)$*, i.e.,* x *s.t.* $(x, x) \in (\mathsf{ghb}'_A(E, X))^{+}$*. In the first two cases* $(*)$*,* ab' *connects two non-fence events, a contradiction. Hence a cycle in* ghb' *implies one in* $\mathsf{r}_1 \cup \mathsf{r}_2$*, i.e., in* ghb *since* $\mathsf{r}_2 \subseteq \mathsf{ab}$*.*

Prefixes: as a contradiction, take an infinite path in $\mathsf{ghb}'_A(E, X)^{+}$*. Only the cases* $(\mathsf{r}_1{}^{+} \cup \mathsf{r}_2{}^{+})^{+}$ *and* $\mathsf{ab}'; (\mathsf{r}_1{}^{+} \cup \mathsf{r}_2{}^{+})^{+}$ *of* $(*)$ *apply, and both imply an infinite path in* $(\mathsf{r}_1{}^{+} \cup \mathsf{r}_2{}^{+})^{+}$*. Hence, we have an infinite prefix in* ghb^{+}*, since* $\mathsf{r}_2 \subseteq \mathsf{ab}$*.*

To decide the satisfiability of ϕ, we can use any solver supporting a sufficiently rich fragment of first-order logic. The procedure reveals the concrete executions, as expressed by Thm. 1.

5.7 Comparison to [14] and [61,62]

Both [14] and [61,62] use an SSA encoding similar to our ssa of Sec. 4. The difference resides in the ordering constraints.

[14] encodes total orders over memory accesses. Thus, in contrast to our clock variables with less-than constraints, [14] uses a Boolean variable M_{xy} per

pair (x, y), whose value places x and y in a total order: either x before y, or y before x. Prog. 1 has $3 \cdot \mathrm{N}$ memory accesses per thread, hence [14]'s encoding has $6 \cdot \mathrm{N} \cdot (6 \cdot \mathrm{N} - 1)$ Boolean variables. [14] builds additional constraints for the transitive closure; their number is at least cubic in the number of variables M_{xy}, leading to $\mathcal{O}(\mathrm{N}^6)$ constraints.

We only consider relations per address, except for program order and fence orderings, and do not build transitive closures. The constraints for fr and ab are cubic in the worst case; all others are quadratic. In Prog. 1, the write serialisation is internal, hence fr is only quadratic. Hence our number of constraints is $\mathcal{O}(\mathrm{N}^2)$.

[61,62] use partial orders like us; they note redundancies in their constraints in [62] but do not explain them, which we do below. Basically, [61,62] quantify over all events regardless of their address, whereas we mostly build constraints per address. Fig. 7 shows that the maximal number of events to a single address is experimentally much smaller than the total number of events.

Our notations correspond to the ones of [62] as follows (the original description [61] has different notations). $\mathrm{HB}(a, b)$ is our clock constraint c_{ab}. The functions addr and val map to ours; $\mathrm{en}(x)$ is our $\mathrm{g}(x)$; $\mathrm{link}(r, w)$ denotes that r reads from w, i.e., our s_{wr}. [62] expresses po, rf, fr, and ws as follows (since it is restricted to SC, it gives no encoding of ppo_A, grf_A, and ab_A).

[62] encodes po as the conjunction of the $c_{a_i a_j}$, with a_i in po before a_j. If the implementation of [62] strictly follows this definition, it redundantly includes the transitive closure constraints, which we avoid by building the transitive reduction in Alg. 4.

[62] encodes rf in $\Pi_1 := \forall r. \exists w.\, \mathrm{g}(r) \Rightarrow (\mathrm{g}(w) \wedge s_{wr})$ and $\Pi_2 := \forall r. \forall w. s_{wr} \Rightarrow (c_{wr} \wedge \mathrm{addr}(r) = \mathrm{addr}(w) \wedge \mathrm{val}(r) = \mathrm{val}(w))$. [62] forces rf to be exclusive. We explained in Sec. 5.1 why this is unnecessary in our case, which allows us to only build a disjunction over writes (cf. Alg. 1) linear in their number.

Π_2 combines our value and clock constraints, with one major difference: Π_2 ranges over all reads and writes, regardless of their address. Our rf (Alg. 1) ranges over pairs to the same address, thus reaches this number only when all reads and writes have the same memory address, which is unlikely in non-trivial programs.

Ranging over the same address, as we do, and not all addresses, as in [62], becomes even more advantageous in Π_3, encoding fr: $\Pi_3 := \forall r. \forall w. \forall w'. (\mathrm{s}_{wr} \Rightarrow (\mathrm{g}(w') \wedge \neg c_{w'w} \wedge \neg c_{rw'} \Rightarrow \mathrm{addr}(r) \neq \mathrm{addr}(w'))$. Π_3 ranges over all (r, w, w'), again independently of their addresses. For distinct addresses the conjunction holds trivially, but [62] builds it nevertheless. Our fr (cf. Alg. 3) quantifies only over the same address, thus spares these trivial constraints.

[62] does not encode ws. The totality of ws comes as a side effect: [62] initialises each write with a unique integer, hence writes are totally ordered by $<$ over integers. This is again regardless of the addresses, whereas we order writes to the same address only.

6 Experimental Results

We detail here our experiments, which indicate that our technique is scalable enough to verify non-trivial, real-world concurrent systems code, including the worker-synchronisation logic of the relational database PostgreSQL, code for socket-handover in the Apache httpd, and the core API of the Read-Copy-Update mutual exclusion code from Linux 3.2.21.

We implement our technique within the bounded model checker CBMC [18], using a SAT solver as an underlying decision procedure. We see two primary comparison points to estimate the overhead introduced by the partial order constraints. First, we pass the benchmarks with a single, fixed interleaving to sequential CBMC. Our implementation performs comparably to sequential CBMC, as Fig. 7 shows (rows "sequential" and "concurrent"). Second, we compare to ESBMC [19], which also implements bounded model checking, but uses interleaving-based techniques.

In Fig. 7, we gather facts about all examples: the Fibonacci example from [11] with N=5, 4500 litmus tests (see below), the worker synchronisation in PostgreSQL, RCU, and fdqueue in Apache httpd. For each we give the number of lines of code (LOC), the number of distinct memory addresses "tot. addr" (including unused shared variables), the total number of shared accesses "tot. shared", the maximal number of accesses to a single address "same addr", the total number of constraints "all constr" and the relation with the most costly encoding, in terms of the number of constraints generated. We give the loop unrolling bounds "unroll": we write "none" when there is no loop, and "bounded" when the loops in the program are natively bounded.

The total number of shared accesses is on average 13 times the maximal number of accesses to a single address. The most costly constraint is usually the read-from, or the barriers, which build on read-from. The time needed by our tool to analyse a program grows with the total number of constraints generated. ESBMC is 4 times slower than our tool on Fibonacci, 3050 times slower on the litmus tests, times out on PostgreSQL, and cannot parse RCU and Apache.

	Fibo.	Litmus	PgSQL	RCU	Apache
LOC	41	50.9	5412	5834	28864
unroll	5	none	2	bounded	5
tot. addr	2	11.8	6	3	8
tot. shared	45	58.7	233	107	88
same addr	11	3.7	72	4	5
all constr	308	874	3762	90	160
most costly	rf (178)	ab (342)	rf (1868)	rf (33)	rf (49)
sequential	0.3 s	0.1 s	4.1 s	0.8 s	1.7 s
concurrent	3.3 s	0.2 s	90.0 s	1.0 s	2.8 s
ESBMC	13.8 s	609.8 s	t/o	parse err	parse err

Fig. 7. Facts about all examples

	CBMC	CBMC	CBMC	CheckFence	CImpact	ESBMC	Poirot	SatAbs	Threader
	SC	TSO	Power	SC, TSO	SC	SC	SC	SC	SC
F	CE $N = 300$	CE $N = 220$	CE $N = 240$	conv err	t/o $N = 1$	CE $N = 10$	fails $N \geq 1$	V $N = 3$	t/o $N = 1$
L	100%	100%	100%	18%	20%	34%	47%	100%	8%
P	V	V	CE	conv err	aborts	t/o	parse err	t/o	n/a
Pf	V	V	V	conv err	aborts	t/o	parse err	t/o	n/a
R	V	V	V	conv err	aborts	parse err	parse err	ref err	n/a
A	V	V	V	conv err	aborts	parse err	parse err	aborts	n/a

Fig. 8. Comparison of all tools on all examples (time out 30 mins)

Other tools There are very few tools for verifying concurrent C programs, even on SC [21]. For weak memory, existing techniques are restricted to TSO, and its siblings PSO and RMO [14,44,43,9,3,49]. Not all of them have been implemented, and only few handle systems code given as C programs.

Thus, as a further comparison point, we implemented an instrumentation technique [7], similar to [9]. The technique of [9] is restricted to TSO, and consists in delaying writes, so that the SC executions of the instrumented code simulate the TSO executions of the original program. Our instrumentation handles all the models of Sec. 3.

We tried 5 ANSI-C model checkers: SatAbs, a verifier based on predicate abstraction [17]; ESBMC; CImpact, a variant of the Impact algorithm [52] extended to SC concurrency; Threader, a thread-modular verifier [34]; and Poirot, which implements a context-bounded translation to sequential programs [45]. These tools cover a broad range of techniques for verifying SC programs. We also tried CheckFence [14].

In Fig. 8, we compare all tools on all examples: F for Prog. 1, L for the litmus tests, P for PostgreSQL with its bug, Pf for our fix, R for RCU and A for Apache. For L, P, R and A, the bounds are as in Fig. 7; for Pf we take the one of P. For F we try the maximal N that the tool can handle within the time out of 30 mins. For each tool, we specify the model below. We write "t/o" when there is a timeout. We write "fail" when the tool gives a wrong answer. CheckFence provides a conversion module from C to its internal representation; we write "conv err" when it fails. We write "parse err" when the tool cannot parse the example. SatAbs uses a refinement procedure; we write "ref err" when it fails. When a tool verifies an example we write "V"; when it finds a counterexample we write "CE".

Fibonacci All tools, except for ESBMC, SatAbs and ours, fail to analyse Fibonacci. Poirot claims the assertion is violated for any N, which is not the case for $1 \leq \text{N} \leq 5$. SatAbs does not reach beyond $\text{N} = 4$. Our tool handles more than $\text{N} = 300$, which is 30 times more loop unrolling than ESBMC, within the same amount of time.

Litmus tests We analyse 4500 tests exposing weak memory artefacts, e.g., instruction reordering, store buffering, store atomicity relaxation. These tests are generated by the diy tool [8], which generates assembly programs with a final state unreachable on SC, but reachable on a weaker model. For example, **iriw**

(Fig. 2) can only be reached on RMO (by reordering the reads) or on Power (*idem*, or because the writes are non-atomic).

We convert these tests into C code, of 50 lines on average, involving 2 to 4 threads. Despite the small size of the tests, they prove challenging to verify, as Fig. 8 shows: most tools, except Blender, SatAbs and ours, give wrong results or fail in other ways on a vast majority of tests, even for SC. For each tool we give the average percentage of correct results over all models. Our tool verifies all tests on all models in 0.22 s on average.

PostgreSQL Developers observed that a regression test failed on a PowerPC machine[4], and later identified the memory model as possible culprit: the processor could delay a write by a thread until after a token signalling the end of this thread's work had been set. Our tool confirmed the bug, and proved a patch we proposed. A detailed description of the problem is in [7].

RCU *Read-Copy-Update* (RCU) is a synchronisation mechanism of the Linux kernel, introduced in version 2.5. Writers to a concurrent data structure prepare a fresh component (e.g., list element), then replace the existing component by adjusting the pointer variable linking to it. Clean-up of the old component is delayed until there is no process reading.

Thus readers can rely on very lightweight (and thus fast) lock-free synchronisation only. The protection of reads against concurrent writes is fence-free on x86, and uses only a light-weight fence (`lwsync`) on Power. We verify the original implementation of the 3.2.21 kernel for x86 (5824 lines) and Power (5834 lines) in less than 1 s, using a harness that asserts that the reader will not obtain an inconsistent version of the component. On Power, removing the `lwsync` makes the assertion fail.

Apache The Apache httpd is the most widely used HTTP server software. It supports a broad range of concurrency APIs distributing incoming requests to a pool of workers.

The fdqueue module (28864 lines) is the central part of this mechanism, which implements the hand-over of a socket together with a memory pool to an idle worker. The implementation uses a central, shared queue for this purpose. Shared access is primarily synchronised by means of an integer keeping track of the number of idle workers, which is updated via architecture-dependent compare-and-swap and atomic decrement operations. Hand-over of the socket and the pool and wake-up of the idle thread is then coordinated by means of a conventional, heavy-weight mutex and a signal. We verify that hand-over guarantees consistency of the payload data passed to the worker in 2.45 s on x86 and 2.8 s on Power.

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

7 Conclusion

Our experiments demonstrate that weakness is a virtue for programs with bounded loops. Our proofs suggest that this contention is not limited to bounded loops, but impracticable as is, since it involves infinite structures. Thus we believe that this work opens up new possibilities for over-approximation for programs with unbounded loops, which we hope to investigate in the future.

Acknowledgements We would like to thank Lihao Liang and Alex Horn for their detailed comments on earlier versions of this paper.

References

1. Sparc Architecture Manual Version 9 (1994)
2. Alpha Architecture Reference Manual, Fourth Edition (2002)
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS (2012)
4. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer (1995)
5. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Princiles, Techniques, and Tools. Addison-Wesley (1986)
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., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation, to appear in ESOP 2013, available at `http://www.cprover.org/etaps/`
8. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models (Extended Version). In: FMSD (2012)
9. Atig, M.F., Bouajjani, A., Parlato, G.: Getting Rid of Store-Buffers in the Analysis of Weak Memory Models. In: CAV (2011)
10. Ben-Asher, Y., Farchi, E.: Using True Concurrency to Model Execution of Parallel Programs. In: IJPP (1994)
11. Beyer, D.: Competition on software verification - (SV-COMP). In: TACAS (2012)
12. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model checking without BDDs. In: TACAS (1999)
13. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS (1990)
14. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
15. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In: CAV (2008)
16. Chambers, B., Manolios, P., Vroon, D.: Faster sat solving with better cnf generation. In: DATE (2009)
17. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS (2005)
18. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004)
19. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE (2011)

20. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. (1991)
21. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. TCAD (2008)
22. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A Race and Transaction-Aware Java Runtime. In: PLDI (2007)
23. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In:POPL 11
24. Flanagan, C., Freund, S., Qadeer, S.: Thread-Modular Verificaton for Shared-Memory Programs. In: ESOP (2002)
25. Flanagan, C., Freund, S., Yi, J.: Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In: PLDI (2008)
26. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model-Checking Software. In: POPL (2005)
27. Flanagan, C., Qadeer, S.: Thread-Modular Model Checking. In: SPIN (2003)
28. Flanagan, C., Qadeer, S., Seshia, S.: A Modular Checker for Multi-Threaded Programs. In: CAV (2002)
29. Ganai, M., Gupta, A.: Efficient Modeling of Concurrent Systems in BMC. In: SPIN (2008)
30. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL 97
31. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996)
32. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An Efficient Execution Verification Tool for Memory Orderings. In: CAV (2004)
33. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV 97
34. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A Constraint-Based Verifier for Multi-Threaded Programs. In: CAV (2011)
35. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. STTT (2000)
36. Holzmann, G.: The model checker SPIN. TOSE (1997)
37. Huynh, Q., Roychoudhury, A.: A memory sensitive checker for C#. In: FM (2006)
38. Jin, H., Yavuz-Kahveci, T., Sanders, B.A.: Java memory model-aware model checking. In: TACAS (2012)
39. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. (1983)
40. Kahloon, V., Wang, C.: Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In: CAV (2010)
41. Kaiser, A., Kroening, D., Wahl, T.: Efficient coverability analysis by proof minimization. In: CONCUR (2012)
42. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC (2003)
43. Kuperstein, M., Vechev, M., Yahav, E.: Partial-Coherence Abstractions for Relaxed Memory Models. In: PLDI (2011)
44. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
45. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: FMSD (2009)
46. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. CACM (1978)

47. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. (1979)
48. Lee, J., Midkiff, S., Padua, D.: Concurrent Static Single Assignment Form and Constant Propagation for Explicit Parallel Programs. In: In PPoPP (1997)
49. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI (2012)
50. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M., Sewell, P., Williams, D.: An Axiomatic Memory Model for Power Multiprocessors. In: CAV (2012)
51. Mazurkiewicz, A.: Basic Notions of Trace Theory. In: REX (1988)
52. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV (2006)
53. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: CAV (1992)
54. Musuvathi, M., Qadeer, S.: Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In: PLDI (2005)
55. Owens, S., Sarkar, S., Sewell, P.: A better x86 model: x86-TSO. In: TPHOL (2009)
56. Peled, D.: All from one, one for all. In: CAV (1993)
57. Plotkin, G., Pratt, V.: Teams can see pomsets. In: POMIV (1996)
58. Pratt, V.: Modeling Concurrency with Partial Orders. In: International Journal of Parallel Programming (1986)
59. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: TACAS (2005)
60. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding Power Multiprocessors. In: PLDI (2011)
61. Sinha, N., Wang, C.: Staged Concurrent Program Analysis. In: FSE (2010)
62. Sinha, N., Wang, C.: On Interference Abstractions. In: POPL (2011)
63. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking Axiomatic Specifications of Memory Models. In: PLDI (2010)
64. Winskel, G.: Event Structures. In: Advances in Petri Nets (1986)