Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

JADE ALGLAVE, Microsoft Research, University College London
DANIEL KROENING, University of Oxford
VINCENT NIMAL, Microsoft Research
DANIEL POETZL, University of Oxford

Modern architectures rely on memory fences to prevent undesired weakenings of memory consistency. As the fences' semantics may be subtle, the automation of their placement is highly desirable. But precise methods for restoring consistency do not scale to deployed systems code. We choose to trade some precision for genuine scalability: our technique is suitable for large code bases. We implement it in our new musketeer tool, and report experiments on more than 700 executables from packages found in Debian GNU/Linux 7.1, including memcached with about 10,000 LoC.

1. INTRODUCTION

Concurrent programs are hard to design and implement, especially when running on multiprocessor architectures. Multiprocessors implement *weak memory models*, which feature, e.g., *instruction reordering* and *store buffering* (both appearing on x86), or *store atomicity relaxation* (a particularity of Power and ARM). Hence, multiprocessors allow more behaviours than Lamport's *Sequential Consistency* (SC) [Lamport 1979], a theoretical model where the execution of a program corresponds to an interleaving of the operations executed by the different threads. This has a dramatic effect on programmers, most of whom learned to program with SC.

Fortunately, architectures provide special *fence* (or *barrier*) instructions to prevent certain behaviours. Yet both the questions of *where* and *how* to insert fences are contentious, as fences are architecture-specific and expensive in terms of runtime.

Attempts at automatically placing fences include Visual Studio 2013, which offers an option to guarantee acquire/release semantics (we study the performance impact of this policy in Section 2). The C++11 standard provides an elaborate API for inter-thread communication, giving the programmer some control over which fences are used, and where. But the use of such APIs might be a hard task, even for expert programmers. For example, Norris and Demsky [2013] reported a bug found in a published C11 implementation of a work-stealing queue.

We address here the question of how to *synthesise* fences, i.e., how to automatically place them in a program to enforce robustness/stability [Bouajjani et al. 2011; Alglave and Maranget 2011], which implies SC. This should lighten the programmer's burden. The fence synthesis tool needs to be based on a precise model of weak memory. In verification, models commonly adopt an *operational* style, where an execution is an interleaving of transitions accessing the memory (as in SC). To address weaker architectures, the models are augmented with buffers and queues that implement the

This work is supported by SRC 2269.002, EPSRC H017585/1 and ERC 280053.

DOI:http://dx.doi.org/10.1145/0000000.0000000

features of the hardware. Similarly, a good fraction of the fence synthesis methods, e.g., Linden and Wolper [2013], Kuperstein et al. [2010], Kuperstein et al. [2011], Liu et al. [2012], Abdulla et al. [2013], and Bouajjani et al. [2013] rely on operational models to describe executions of programs.

Challenges. Methods using operational models inherit the limitations of methods based on interleavings, e.g., the *"severely limited scalability"*, as Liu et al. [2012] put it. Indeed, none of them scale to programs with more than a few hundred lines of code, due to the very large number of executions a program can have. Another impediment to scalability is that these methods establish if there is a need for fences by exploring the executions of a program one by one.

Finally, considering models like Power or ARM makes the problem significantly more difficult. Intel x86 offers only one fence (mfence), but Power offers a variety of synchronisation mechanisms: fences (e.g., sync and lwsync) and dependencies (address, data, or control). This diversity makes the optimisation more subtle: one cannot simply minimise the number of fences, but rather has to consider the costs of the different synchronisation mechanisms; for instance, it might be cheaper to use one full fence than four dependencies.

Our approach. We tackle these challenges with a static approach. Our choice of model almost mandates this approach: we rely on the axiomatic semantics of Alglave et al. [2010]. We feel that an axiomatic semantics is an invitation to build abstract objects that embrace all the executions of a program.

Previous works, e.g., [Shasha and Snir 1988; Alglave and Maranget 2011; Bouajjani et al. 2011; Bouajjani et al. 2013], show that weak memory behaviours boil down to the presence of certain cycles, called *critical cycles*, in the executions of the program. A critical cycle essentially represents a minimal violation of SC, and thus indicates where to place fences to restore SC. We detect these cycles statically, by exploring an over-approximation of the executions of the program.

Contributions. We describe below the contributions of our paper:

— A self-contained introduction to axiomatic memory models, including a detailed account of the special shapes of critical cycles (Section 4).
— A fence inference approach, based on finding critical cycles in the abstract event graph (aeg) of a program (Section 5), and then computing via a novel integer linear programming formulation a minimal set of fences to guarantee sequential consistency (Section 6). The approach takes into account the different costs of fences, and is sound for a wide range of architectures, including x86-TSO, Power, and ARM.
— The first formal description of the construction of aegs (Section 5.4), and a correctness proof showing that the aeg does capture all potential executions of the analysed program (Section 5.5). This includes a description of how to correctly use overapproximate points-to information during the construction of the aeg. The aeg abstraction is not specific to fence insertion and can also be used for other program analysis tasks.
— A formalisation of the generation of the event structures and candidate executions of a program in the framework of Alglave et al. [2010] (Section 5.5). This has in previous work only been treated informally.
— An implementation of our approach in the new tool musketeer and an evaluation and comparison of our tool to others (Sections 2 and 7). Our evaluation on both classic examples (such as Dekker's algorithm) and large real-world programs from the Debian GNU/Linux distribution (such as memcached which has about 10,000 LoC) shows that our method achieves good precision and scales well.

— A study of the performance impact of fences inserted by different fence insertion methods (Sections 2 and 7.3). For this study we implemented several of the competing approaches (such as pensieve).
— A description of how to insert fences into a C program using inline assembly and taking into account data, address, and control dependencies (Section 7.1).

Outline. We discuss the performance impact of fences in Section 2, and survey related work in Section 3. We give an introduction to axiomatic memory models in Section 4. We detail how we detect critical cycles in Section 5, and how we place fences in Section 6. In Section 7, we report on an experimental comparison between the existing methods and our new tool musketeer. We provide full sources, benchmarks and experimental reports online at http://www.cprover.org/wmm/musketeer.

2. MOTIVATION

Before considering elaborate methods for the placement of fences, we investigated whether naive approaches to fence insertion indeed have a negative performance impact.

There is surprisingly little related work; we found [Sura et al. 2005; Marino et al. 2011; Spear et al. 2009; Fang et al. 2003]. Nevertheless, fences are considered to be amongst the most expensive instructions: Herlihy and Shavit [2008] write that "*memory barriers are expensive (100s of cycles, maybe more), and should be used only when necessary*." Bouajjani et al. [2011] benchmarked x86's mfence (see http://concurrency.informatik.uni-kl.de/trencher.html), and observe that it has a significant cost when used in isolation. Similar observations were made by Alglave and Maranget [2011] (also see http://offence.inria.fr/exp/speed.html).

We measured the overhead of different fencing methods on a stack and a queue from the liblfds lock-free data structure package (http://liblfds.org). For each data structure, we create a harness (consisting of 4 threads) that concurrently invokes its operations.

We built several versions of the above two programs:

— (M) with fences inserted by our tool musketeer;
— (P) with fences following the *delay set analysis* of the pensieve compiler [Sura et al. 2005], i.e., a static over-approximation of Shasha and Snir's eponymous (dynamic) analysis [Shasha and Snir 1988];
— (V) with fences following the *Visual Studio* policy, i.e., guaranteeing acquire/release semantics (in the C11 sense [C11 2011]) for reads and writes of `volatile` variables (see http://msdn.microsoft.com/en-us/library/vstudio/jj635841.aspx, accessed 04-11-2013). We emphasise that this method does not guarantee SC, and we include it only as a comparison point here. On x86, no fences are necessary to enforce acquire/release semantics, as the model is sufficiently strong already. Hence, we only provide data for ARM.
— (E) with fences after each access to a shared variable;
— (H) with an mfence (x86) or a dmb (ARM) after every assembly instruction that writes (x86) or reads or writes (ARM) any global or heap-allocated data.

These experiments required us to implement (P), (E), and (V) ourselves, so that they would handle the architectures that we considered. This means in particular that our tool provides the pensieve policy (P) for TSO, Power, and ARM, whereas the original pensieve targeted Java only.

We compiled all the program versions, i.e., both the original program and the program with fences inserted according to the different fencing strategies, with `gcc -O0`. We ran all versions 100 times on an x86-64 Intel Core i5-3570 with 4 cores at 3.40 GHz and

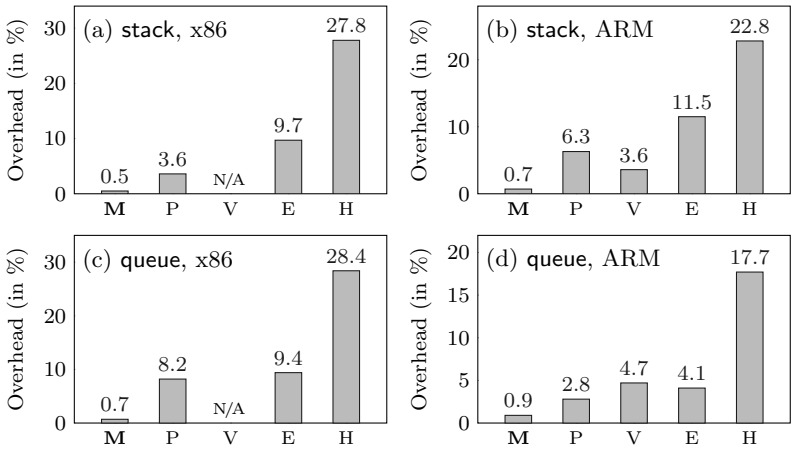


Fig. 1. Overheads for the different fencing strategies.

	stack on x86	stack on ARM	queue on x86	queue on ARM
(M)	[10.059; 10.089]	[11.950; 11.973]	[12.296; 12.328]	[21.419; 21.452]
(P)	[10.371; 10.400]	[12.608; 12.638]	[13.206; 13.241]	[21.818; 21.850]
(V)	N/A	[12.294; 12.318]	N/A	[22.219; 22.255]
(E)	[10.989; 11.010]	[13.214; 13.259]	[13.357; 13.390]	[22.099; 22.128]
(H)	[12.788; 12.838]	[14.574; 14.587]	[15.686; 15.723]	[24.983; 25.013]

Fig. 2. Confidence intervals for the mean execution times (in sec) for the data structure experiments.

4 GB of RAM, and on an ARMv7 (32-bit) Samsung Exynos 4412 with 4 cores at 1.6 GHz and 2 GB of RAM.

For each program version, Figure 1 gives the mean overhead w.r.t. the unfenced program. We give the overhead (in %) in *user time* (as given by Linux `time`), i.e., the time spent by the program in user mode on the CPU. Amongst the approaches that guarantee SC (i.e., all but V), the best results were achieved with our tool musketeer.

We checked the statistical significance of the execution time improvement of our method over the existing methods by computing and comparing the confidence intervals for the mean execution times. The sample size is $N = 100$ and the confidence level is $1 - \alpha = 95\%$. The confidence intervals are given in Figure 2. If the confidence intervals for two methods are non-overlapping, we can conclude that the difference between the means is statistically significant.

As discussed later in Section 5.1, the fence insertion approaches compared in this section analyse C programs while assuming a straightforward compilation scheme to assembly in which accesses are not reordered or otherwise optimised by the compiler. Thus, sound results are only achieved when using compilation settings that guarantee these properties (e.g., `gcc -O0`).

Nevertheless, we also compared the approaches when compiling with `-O1` to get an estimate about how the different approaches would fare when allowing more compiler optimisations. We observed that on x86 the runtime decreased between 1% and 9%. On ARM the runtime decreased between 3% and 31%. The relative performance of the different approaches remained the same as with `-O0`, i.e., the best runtime was achieved with musketeer (M) while the approach H (fence after every access to static or heap memory) was slowest. We give the corresponding data online at http://www.cprover.org/wmm/musketeer.

authors	tool	model style	objective
Abdulla et al. [2013]	memorax	operational	reachability
Alglave et al. [2010]	offence	axiomatic	SC
Bouajjani et al. [2013]	trencher	operational	SC
Fang et al. [2003]	pensieve	axiomatic	SC
Kuperstein et al. [2010]	fender	operational	reachability
Kuperstein et al. [2011]	blender	operational	reachability
Linden and Wolper [2013]	remmex	operational	reachability
Liu et al. [2012]	dfence	operational	specification
Sura et al. [2005]	pensieve	axiomatic	SC
Abdulla et al. [2015]	persist	operational	persistence

Fig. 3. Overview of existing fence synthesis tools.

3. RELATED WORK

The work of Shasha and Snir [1988] is a foundation for much of the field of fence synthesis. Most of the work cited below inherits their notions of *delay* and *critical cycle*. A delay is a pair of instructions in a thread that can be reordered by the underlying architecture. A critical cycle essentially represents a minimal violation of sequential consistency.

Figure 3 classifies the methods that we compare to w.r.t. their style of model (operational or axiomatic). The table further indicates the objective of the fence insertion procedure: enforcing SC, preventing reachability of error states (i.e., ensuring safety properties), or other specifications (such as enforcing given orderings of memory accesses).

We report on our experimental comparison to these tools in Section 7. We now summarise the fence synthesis methods per style. We write TSO for Total Store Order, implemented in Sparc TSO [SPARC 1994] and Intel x86 [Owens et al. 2009]. We write PSO for Partial Store Order and RMO for Relaxed Memory Order, two other Sparc architectures. We write Power for IBM Power [Power 2009].

3.1. Operational models

Linden and Wolper [2013] explore all executions (using what they call *automata acceleration*) to simulate the reorderings occuring under TSO and PSO. Abdulla et al. [2013] couple predicate abstraction for TSO with a counterexample-guided strategy. They check if an error state is reachable; if so, they calculate what they call the *maximal permissive* sets of fences that forbid this error state. Their method guarantees that the fences they find are *necessary*, i.e., removing a fence from the set would make the error state reachable again. A precise method for PSO is presented by Abdulla et al. [2015].

Kuperstein et al. [2010] explore all executions for TSO, PSO and a subset of RMO, and along the way build constraints encoding reorderings leading to error states. The fences can be derived from the set of constraints at the error states. The same authors [Kuperstein et al. 2011] improve this exploration under TSO and PSO using an abstract interpretation they call *partial coherence abstraction*, relaxing the order in the write buffers after a certain bound, thus reducing the state space to explore. Meshman et al. [2014] synthesise fences for infinite-state algorithms to satisfy safety specifications under TSO and PSO. The approach works by *refinement propagation*: They successively refine the set of inferred fences by combining abstraction refinements of the analysed program. Liu et al. [2012] offer a *dynamic synthesis* approach for TSO and PSO, enumerating the possible sets of fences to prevent an execution picked dynamically from reaching an error state.

Bouajjani et al. [2013] build on an operational model of TSO. They look for *minimum violations* (viz. critical cycles) by enumerating *attackers* (viz. delays). Like us, they use integer linear programming (ILP). However, they first enumerate all the solutions, then encode them as an ILP, and finally ask the solver to pick the least expensive one. Our method directly encodes the whole decision problem as an ILP. The solver thus both constructs the solution (avoiding the exponential-size ILP problem) and ensures its optimality.

Abdulla et al. [2015] investigate a new property called *persistence* as a comprise between optimality and efficiency. A TSO program is persistent if, for any trace t, there exists an SC trace t' in which the program order and store order between events are identical to those in t. If a program is not persistent, the tool persist uses patterns to find *fragility* – the cause of non-persistence – and infers a set of fences.

All the approaches above focus on TSO and its siblings PSO and RMO, whereas we also handle the significantly weaker Power model, including subtle barriers (e.g., lwsync), compared to the simpler mfence of x86.

3.2. Axiomatic models

Krishnamurthy and Yelick [1996] apply Shasha and Snir's method to *single program multiple data* systems. Their abstraction is similar to ours, except that they do not handle pointers. Moreover, since their programs are symmetrical, their abstraction can be much smaller than in the general case.

Lee and Padua [2001] propose an algorithm based on Shasha and Snir's work. They use dominators in graphs to determine which fences are redundant. This approach was later implemented by Fang et al. [2003] in pensieve, a compiler for Java. Sura et al. later implemented a more precise approach in pensieve [Sura et al. 2005] (see (P) in Section 2). They pair the cycle detection with an analysis to detect synchronisation that could prevent cycles.

Alglave et al. [2010] revisit Shasha and Snir for contemporary memory models and insert fences following a refinement of [Lee and Padua 2001]. Their offence tool handles snippets of assembly code only, where the memory locations need to be explicitly given.

3.3. Others

The work of Vafeiadis and Zappa Nardelli [2011] presents an optimisation of the certified CompCert-TSO compiler to remove redundant fences on TSO.

Marino et al. [2011] experiment with an SC-preserving compiler, showing overheads of no more than $34\,\%$. Nevertheless, they state that *"the overheads, however small, might be unacceptable for certain applications"*.

Bender et al. [2015] provide a fence insertion approach to enforce a set of declared orderings between memory accesses. They model the fence synthesis problem as a minimum multi-cut problem on the control flow graph, and determine the set of fences necessary to additionally enforce the orders that are not already enforced by the architecture (in the absence of fences).

Joshi and Kroening [2015] provide a fence synthesis approach based on bounded model checking that aims at deducing a set of fences sufficient to guarantee the assertions in the program. They iteratively consider counterexamples of increasing length, and reduce the problem of finding a set of fences to computing the minimum hitting set over a set of reorderings.

Lustig et al. [2015] describe a dynamic fence insertion approach. Based on a description of the memory model of the source and target architecture, they generate a finite state machine that dynamically translates code compiled for the source memory model to correctly execute on hardware implementing the target memory model.

4. AXIOMATIC MEMORY MODELS

Weak memory effects can occur as follows: a thread sends a write to a store buffer, then to a cache, and finally to memory. While the write transits through buffers and caches, reads can occur before the written value is available in memory to all threads.

To describe such situations, we use the framework of Alglave et al. [2010], embracing in particular SC, Sun TSO (i.e., the x86 model [Owens et al. 2009]), Power, and ARM. In this framework, a memory model is specified as a predicate on *candidate executions*. The predicate indicates whether a candidate execution is allowed (i.e., may occur) or disallowed (i.e., cannot occur) on the respective architecture. A candidate execution is represented as a directed graph. The nodes are memory events (reads or writes), and the edges indicate certain relations between the events. For example, a read-from (rf) edge from a write to a read indicates that the read takes its value from that write.

We illustrate this framework in the next section using a *litmus test* (Figure 4). A litmus test is a short concurrent program together with a condition on its final state. The given litmus test consists of two threads, which access shared variables x and y. The shared variables are assumed to be initialized to zero at the beginning. The given condition holds for an execution in which load (c) reads value 1 from y, and load (d) reads value 0 from x. Whether the given litmus test has an execution that can end up in this final state depends on the memory model. For example, the given outcome can occur on Power but not on TSO. Thus, for a given architecture, a set of litmus tests together with the information of whether the given outcome can occur on any execution characterises the architecture.

4.1. Basics

We next describe how the set of candidate executions of a program is defined. A candidate execution is obtained by first generating an *event structure*. An event structure $E \triangleq (\mathbb{E}, \mathsf{po})$ is a set of memory events $\mathbb{E}$ together with the program order relation po.[1] An *event* is a read from memory or a write to memory, consisting of an identifier, a direction (R for read or W for write), a memory address (represented by a variable name) and a value. The program order po is a per-thread total order over $\mathbb{E}$. An event structure represents an execution of the program, assuming the shared reads can return *arbitrary* values.

For example, Figure 5a gives an event structure associated with the litmus test in Figure 4. A store instruction (e.g., $\mathtt{x} \leftarrow \mathtt{1}$ on T_0) gives rise to a write event (e.g., (a)Wx1), and a load instruction (e.g., $\mathtt{r1} \leftarrow \mathtt{y}$ on T_1) gives rise to a read event (e.g., (c)Ry1). In this particular event structure, we have assumed that the load (c) on T_1 read value 1, and the load (d) on T_1 read value 0, but any value for the loads (c) and (d) would give rise to a valid event structure.

An event structure can be completed to a candidate execution by adding an *execution witness* $X \triangleq (\mathsf{co}, \mathsf{rf}, \mathsf{fr})$. An execution witness represents the *communication* between the threads and consists of the three relations co, rf, and fr. The *coherence* relation co is a per-address total order on write events, and 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. The *read-from* relation rf links a write w to a read r such that r reads the value written by w. The fr relation is defined in terms of rf and co (hence we say it is a derived relation). A read r is in fr with a write w if the write w' from which r reads hits the memory before w. Formally, we have: $(r, w) \in \mathsf{fr} \triangleq \exists w'.(w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{co}$.

[1] Our notion of event structures differs from the one previously introduced by Winskel [1986]. Winskel's event structures also contain a conflict relation in addition to a set of events and a partial order over them.

mp

T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `r1` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `r2` $\leftarrow$ `x`

Final state? `r1=1` $\wedge$ `r2=0`

Fig. 4. Message Passing (**mp**).

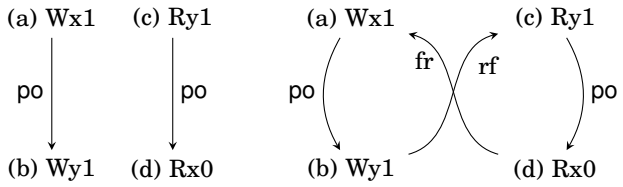


Fig. 5. Event structure and candidate execution

Figure 5b shows the event structure of Figure 5a completed to a candidate execution. A candidate execution is uniquely identified by the event structure E and execution witness X. Inserting fences into a program does not change the set of its candidate executions. For example, if we would place a fence between the two stores of T_0 in Figure 4, the litmus test would still have the same set of candidate executions. However, the fences do affect which of those candidate executions are possible on a given architecture.

Not all event structures can be completed to a candidate execution. For example, had we assumed that the first read in Figure 4 reads value 2, then there would be no execution witness such that the read can be matched up via rf with a corresponding write writing the same value (as there is no instruction writing value 2).

As we have mentioned earlier, a memory model is specified as a predicate on candidate executions. Such a predicate is typically formulated as an acyclicity condition on candidate executions. For example, a candidate execution (E, X) is allowed (i.e., possible) on SC if and only if $\mathsf{acyclic}(\mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr})$. This means that a candidate execution is not possible on SC if it contains at least one cycle formed of edges from po, co, rf, and fr. Consider for example the candidate execution in Figure 5b. This execution is not possible on SC as it has a cycle.

4.2. Minimal cycles

Any execution that has a cycle (i.e., a cycle formed of edges in $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr}$) also has a *minimal cycle*. Given a candidate execution (E, X), a minimal cycle is a cycle such that

> (MC1) per thread, there are at most two accesses, and the accesses are adjacent in the cycle; and
>
> (MC2) for a memory location ℓ, there are at most three accesses to ℓ along the cycle, and the accesses are adjacent in the cycle.

The reason for (MC1) is that the po relation is transitive. That is, given a cycle with more than two accesses for a thread, the po edge from the first to the last access (according to po) forms a chord in the cycle. This chord can be used to bypass the other accesses from the thread and thus form a smaller cycle containing only two accesses of the thread.

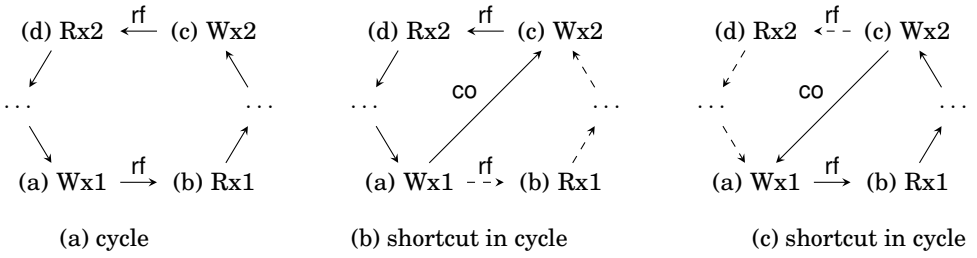


Fig. 6. A cycle in a candidate execution, and two possible shortcuts one can take to form a smaller cycle

The property (MC2) can be seen by careful inspection of the different shapes candidate executions can have, and the shortcuts one can consequently take in cycles. Consider for example Figure 6. Figure 6a shows a cycle involving four accesses to variable x. We know that any two writes to a given variable are connected by a co edge (by the definition of co). Therefore, there either is a co edge from event (a) to event (c) (Figure 6b) or there is a co edge from event (c) to event (a) (Figure 6c). In both cases, we obtain a smaller cycle bypassing one of the accesses to variable x.

We will later extend the notion of minimal cycles to critical cycles, which embody minimal violations of SC (while not violating memory coherence).

4.3. Models weaker than SC

In the axiomatic framework of Alglave et al. [2010], a memory model is specified as a predicate on candidate executions. The predicate is phrased as an acyclicity check on a subrelation of $\mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$. This means that only those executions for which this acyclicity predicate holds are possible on the respective memory model.

The SC model is defined by requiring all relations to be acyclic, i.e., $\mathrm{acyclic}(\mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr})$. In models weaker than SC, only a proper subrelation of $\mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ is required to be acyclic. For a given candidate execution and memory model, we say that those edges which are in $\mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ but not in the subrelation are *relaxed*. In the Power model, for example, the program order edges po between events not separated by a fence are not part of the subrelation of $\mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ required to be acyclic. Hence, the candidate execution depicted in Figure 5b is possible on Power. We thus say that on Power the program order edges are relaxed.

We define in the following a few further relations over memory events that allow us to specify the edges that are relaxed in a model. All these relations are subrelations of the relations already defined. We write dp (with $\mathsf{dp} \subseteq \mathsf{po}$) for the relation that models *dependencies* between instructions. For instance, there is a *data dependency* between a load and a store in an execution when the value written by the store was computed from the value obtained by the load. We further write rfe (resp. coe, fre) for the *external read-from* (resp. external coherence, external from-read), i.e., when the source and target belong to different threads. We write rfi for the *internal read-from*, i.e., when the source and target belong to the same thread. The fence relations (such as $\mathsf{mfence} \subseteq \mathsf{po}$) model architecture-specific fences. They connect all events that occur before the fence to all events that occur after the fence.

Relaxed or safe. When a thread can read from its own store buffer [Adve and Gharachorloo 1995] (the typical TSO/x86 scenario), we relax the internal read-from rfi, that is, rf where source and target belong to the same thread. When two threads T_0 and T_1 can communicate privately via a cache (a case of *write atomicity* relaxation [Adve and Gharachorloo 1995]), we relax the external read-from rfe, and call the corresponding write *non-atomic*. This is the main particularity of Power and ARM, and cannot happen

	SC	x86	Power
poWR	always	mfence	sync
poWW	always	always	sync or lwsync
poRW	always	always	sync or lwsync or dp
poRR	always	always	sync or lwsync or dp or branch;isync

Fig. 7. ppo and fences per architecture.

on TSO/x86. Some program order pairs may be relaxed as well (e.g., write-read pairs on x86), i.e., only a subset of po is guaranteed to occur in order. We call this subset the *preserved program order*, ppo. When a relation is not relaxed on a given architecture, we call it *safe*.

Figure 7 summarises ppo per architecture. The columns are architectures, e.g., x86, and the lines are relations, e.g., poWR. We write, e.g., poWR for the program order between a write and a read. We write "always" when the relation is in the ppo of the architecture: e.g., poWR is in the ppo of SC. When we write something else, typically the name of a fence, e.g., mfence, the relation is not in the ppo of the architecture (e.g., poWR is not in the ppo of x86), and the fence can restore the ordering: e.g., mfence maintains write-read pairs in program order.

Following Alglave et al. [2010], the relation fence (with $\mathsf{fence} \subseteq \mathsf{po}$; for some concrete architecture-specific fence) induced by a fence is *non-cumulative* when it only orders certain pairs of events surrounding the fence. The relation fence is *cumulative* when it additionally makes writes atomic, e.g., by flushing caches. In our model, this amounts to making sequences of external read-from and fences (rfe; fence or fence; rfe) safe, but rfe alone would not be safe. In Figure 4, placing a cumulative fence between the two writes on T_0 will not only prevent their reordering, but also enforce an ordering between the write (a) on T_0 and the read (c) on T_1, which reads from T_0 (in Figure 5b).

Architectures. An *architecture* A determines the relations safe (i.e., not relaxed) on A. We always consider the coherence co, the from-read relation fr and the fence relations to be safe. SC relaxes nothing, i.e., also rf and po are safe. For example, TSO authorises the reordering of write-read pairs (relaxing po edges from a write event to a read event, i.e., poWR) and store buffering (relaxing rfi edges). Thus, the TSO memory model can be phrased as $\mathrm{acyclic}((\mathsf{po} \setminus \mathsf{poWR}) \cup \mathsf{mfence} \cup \mathsf{co} \cup \mathsf{rfe} \cup \mathsf{fr})$. We refer to Alglave et al. [2014] for a description of the Power memory model.

All models we handle satisfy the *SC per location* property. That is, the edges that are part of cycles that consist of events that access only a single memory location are never relaxed. Formally, we have $\mathsf{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr})$, with po-loc restricted to the po edges between events that access the same memory location. This property models the memory coherence provided by modern architectures. We illustrate it with a litmus test in Figure 8. The two threads access the memory location x, which is assumed to be 0 at the beginning. The condition models whether it is possible for T_0 to first read the new value 1, and then read the old value 0. The corresponding candidate execution is depicted on the right. The execution has a cycle that consists of only one memory location. Therefore, this execution is not possible as the edges in such cycles are never relaxed.

4.4. Critical cycles

Following [Shasha and Snir 1988; Alglave and Maranget 2011], for an architecture A, a *delay* is a po or rf edge that is not safe (i.e., is relaxed) on A. A candidate execution (E, X) is valid on A yet not on SC iff

(DC1) it contains at least one cycle that contains a delay, and

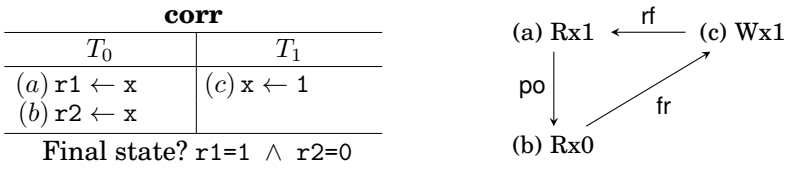


Fig. 8. Read-read coherence.

(DC2) all cycles in it contain a delay.

If there would be a cycle that does not contain a delay, then the execution would be invalid both on SC and A. If there would be no cycles at all in the execution, then the execution would be valid on both SC and A.

To enforce SC on a weaker architecture A, we need to insert memory fences into the program such as to disallow the candidate executions that satisfy properties DC1 and DC2. That is, we need to insert fences such that for each such candidate execution *at least one* cycle is not relaxed. It is not necessary to disallow *all* cycles in a candidate execution, as ensuring that one cycle is not relaxed is sufficient to disallow an execution.

Critical cycles. Any candidate execution that satisfies properties DC1 and DC2 has a cycle, and thus also has a minimal cycle (see Section 4.2). By DC2, the minimal cycle also contains a delay. We refer to such cycles as *critical cycles*. Formally, a critical cycle for architecture A is a cycle which has the following characteristics:

(CS1) the cycle contains at least one delay for A;

(CS2) per thread, there are at most two accesses, the accesses are adjacent in the cycle, and the accesses are to different memory locations; and

(CS3) for a memory location ℓ, there are at most three accesses to ℓ along the cycle, the accesses are adjacent in the cycle, and the accesses are from different threads.

Thus, a critical cycle is a minimal cycle for which it *additionally* holds that (a) it has at least one delay, (b) the accesses of a single thread are to different memory locations, and (c) the accesses to a given location ℓ are from different threads. In fact, together with the properties MC1 and MC2 of minimal cycles, property (a) implies properties (b) and (c), as we show in the next two paragraphs.

To see that (b) holds, assume we have a minimal cycle for which (a) holds but not (b). Thus, there is a thread in the cycle for which its two accesses are to the same location. Then either (i) there is no additional access in the cycle, or (ii) there is an additional access in the cycle. (i) We have a cycle of length 2. This cycle must involve the two accesses (which are to the same memory location). Such cycles are never relaxed due to memory coherence. The cycle thus does not contain a delay. But this contradicts the initial assumption that the cycle has a delay. Therefore, (b) must hold. (ii) By MC2, this location can occur at most three times in the cycle. The third access must be by a different thread. Since communication edges are always between events operating on the same memory location, it follows that we have a cycle of length 3 that mentions only one memory location (cf. Figure 8). Since these cycles are never relaxed, it follows that the cycle does not have a delay. But this contradicts the initial assumption that the cycle has a delay. Therefore, (b) must hold.

We see that property (c) holds since (b) states that a thread accesses different memory locations. Hence, all accesses to a given location must come from different threads.

```
void thread_1(int input)
{
  int r1;
  x = input;
  if (rand()%2)
    y = 1;
  else
    r1 = z;
  x = 1;
}
```

```
void thread_2()
{
  int r2, r3, r4;
  r2 = y;
  r3 = z;
  r4 = x;
}
```

```
thread_1
    int r1;
    x = input;
    _Bool tmp;
    tmp = rand();
    [!tmp%2] goto 1;
    y = 1;
    goto 2;
1:  r1 = z;
2:  x = 1;
    end_function
```

```
thread_2
    int r2, r3, r4;
    r2 = y;
    r3 = z;
    r4 = x;
    end_function
```

Fig. 9. A C program (left) and its goto-program (right).

As an example, the execution in Figure 5b contains a critical cycle w.r.t. Power, formed by the sequence of edges po, rf, po, fr. The po edge on T_0, the po edge on T_1, and the rf edge between T_0 and T_1 are all relaxed on Power. On the other hand, the cycle does not contain an edge that is relaxed on TSO, and it is thus not a critical cycle on TSO.

To forbid executions containing critical cycles, and consequently to enforce SC, one can insert fences into the program to prevent the delays that are part of the cycles. To prevent a po delay, a fence can be inserted between the two accesses forming the delay, following Figure 7. To prevent an rf delay, a cumulative fence must be used (see Section 6 for details). For the example in Figure 4, for Power, we need to place a cumulative fence between the two writes on T_0, preventing both the po and the adjacent rf edge from being relaxed, and use a dependency or fence to prevent the po edge on T_1 from being relaxed.

5. STATIC DETECTION OF CRITICAL CYCLES

We want to synthesise fences to prevent weak behaviours and thus restore SC. As we explained in Section 4, this can be achieved by placing fences such as to prevent the delays along critical cycles of the candidate executions. However, enumerating all candidate executions and looking for critical cycles in each of them separately would not scale beyond small, simple programs.

Therefore, we look for cycles in an over-approximation of all the candidate executions of the program. We hence avoid enumeration of all candidate executions, which would hinder scalability, and get all the critical cycles of all program executions at once. Thus, for example, we can find all fences preventing the critical cycles occuring in two different executions in one step, instead of having to examine the two executions separately.

5.1. Abstract event graphs

We analyse concurrent C programs and assign to them the semantics of the underlying hardware memory model. We thus assume a compilation scheme to assembly in which memory accesses are not reordered, introduced, or removed. Our approach is sound when using compilation settings that guarantee these properties, such as `-O0` with gcc. To analyse a C program, e.g., as given on the left-hand side of Figure 9, we convert it to a *goto-program* (right-hand side of Figure 9), the internal representation of the CProver framework. A goto-program is a sequence of *goto-instructions*, and closely mirrors the C program from which it was generated. We refer to http://www.cprover.org/goto-cc for further details.

The C program in Figure 9 features two threads which can interfere. The first thread writes the argument "input" to x, then randomly writes 1 to y or reads z, and then writes 1 to x. The second thread successively reads y, z and x. In the corresponding goto-program, the **if-else** structure has been transformed into a guard with the condition

of the **if** followed by a goto construct. From the goto-program, we then compute an *abstract event graph* (aeg), given in Figure 10(a). The events a, b_1, b_2 and c (resp. d, e and f) correspond to thread$_1$ (resp. thread$_2$) in Figure 9. We only consider accesses to shared variables, and ignore the local variables. We finally explore the aeg to find the potential critical cycles.

An aeg represents all the candidate executions of a program (in the sense of Section 4). Figure 10(b) and (c) give two executions associated with the aeg given in Figure 10(a). For readability, the transitive po edges have been omitted (e.g., between the two events d' and f'). The concrete events that occur in an execution are shown in bold. In an aeg, the events do not have concrete values, whereas in an execution they do. Also, an aeg merely indicates that two accesses to the same variable could form a data race (see the competing pairs (cmp) relation in Figure 10(a), which is a symmetric relation), whereas an execution has oriented relations (e.g., indicating the write that a read takes its value from, see e.g., the rf arrow in Figure 10(b) and (c)). The execution in Figure 10(b) has a critical cycle (with respect to, e.g., Power) between the events a', b'_2, d', and f'. The execution in Figure 10(c) does not have a critical cycle.

We build an aeg essentially as in [Alglave et al. 2013]. However, our goal and theirs differ: they instrument an input program to reuse SC verification tools to perform weak memory verification, whereas we are interested in automatic fence placement. Moreover, the work of [Alglave et al. 2013] did not present a semantics of goto-programs in terms of aegs, or a proof that the aeg does encompass all potential executions of the program, both of which we do in this section.

5.2. Points-to information

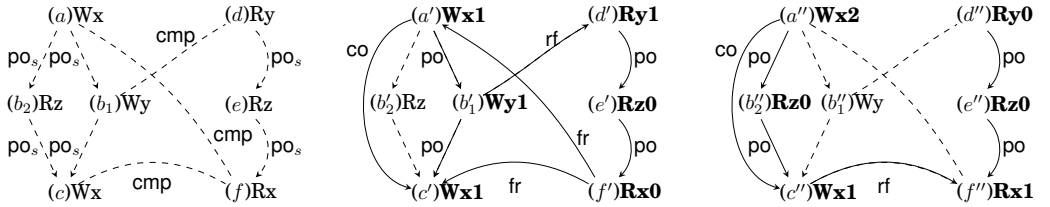
In the previous section, we have denoted abstract events as, e.g., (a)Wx, with x being an address specifier denoting a shared variable in the program. However, shared memory accesses are often performed via pointer expressions. We thus use a pointer analysis to compute which memory locations an expression in the program (such as `a[i+1]` or `*p`) might access. The pointer analysis we use is a standard concurrent points-to analysis that we have shown to be sound for our weak memory models in earlier work [Alglave et al. 2011].[2] The analysis computes for each memory access expression in the goto-program an abstraction of the set of memory locations potentially accessed.

The result of the pointer analysis is for each memory access expression either a set of address specifiers $\{s_1, \ldots, s_n\}$ (denoting that the expression might access any of the memory locations associated with the specifiers, with $s_i \neq *$), or the singleton set $\{*\}$ containing the special address specifier $*$ (denoting that the expression might access *any* memory location). An address specifier $s_i \neq *$ might refer to a single memory location or a region of memory locations. A specifier that refers to a region of memory is for example returned for accesses to arrays. That is, expressions `a[i]` and `a[j]` accessing a global array `a`, with $i \neq j$, would both be mapped to the same specifier `a`, denoting the region of memory associated with the array `a`.

We say that a concrete memory location m and an address specifier s are *compatible*, written comp(m, s), if m is in the set of memory locations abstracted over by s. For example, comp$(m, *)$ holds for any memory location m. As another example, if m refers to a location in array `a`, and specifier s represents that array, then comp(m, s).

Given two address specifiers s_1, s_2, we similarly write comp(s_1, s_2) when the intersection between the set of memory locations abstracted by s_1 and the set of memory

[2]As pointed out by a reviewer, for our fence insertion approach it may be sufficient to use a pointer analysis that is sound for SC but not for weaker models. While this is not the case for all memory models that could be expressed in the framework of Alglave et al. [2010], it may hold for the models we consider in this paper. A proof of this conjecture remains as future work.



(a) aeg of Figure 9 (b) ex. with critical cycle (c) ex. without critical cycle

Fig. 10. The aeg corresponding to the program in Figure 9 and two executions corresponding to it.

locations abstracted by s_2 is non-empty. For example, comp($s_1, *$) holds for any address specifier s_1.

Consequently, instead of being associated with a concrete memory location, abstract events have a single address specifier s_i (which can also be $*$). Thus, if the pointer analysis yields a set of address specifiers $\{s_1, \ldots, s_n\}$ for an expression, the aeg will contain a static event for each. We provide more details in the following sections.

5.3. Formal Definition of Abstract Event Graphs

Given a goto-program (such as the one on the right of Figure 9), we build an aeg $\triangleq (\mathbb{E}_s, \mathsf{po}_s, \mathsf{cmp})$, where $\mathbb{E}_s$ is the set of *abstract events*, po_s is the *static program order*, and cmp are the *competing pairs*. Given an aeg G, we write respectively $G.\mathbb{E}_s$, $G.\mathsf{po}_s$ and $G.\,\mathsf{cmp}$ for the abstract events, the static program order and the competing pairs of G. The aeg for the program on the right of Figure 9 is given in Figure 10(a).

Abstract events. An abstract event represents all events with same program point, direction (write or read), and compatible memory location. An abstract event consists of first a unique identifier, then the direction specifier (W or R), and then the address specifier. In Figure 10(a), (a)Wx abstracts the events (a')Wx1 and (a'')Wx2 in the executions of Figure 10(b) and (c). Moreover, for example, a static event (a)W$*$ would also abstract the two events, as $*$ is compatible with any memory location. We write addr(e) for the address specifier of an abstract event e.

Static program order. The static program order relation po_s abstracts all the (dynamic) po edges that connect two events in program order and that cannot be decomposed as a succession of po edges in this execution. We write po_s^+ (resp. po_s^*) for the transitive (resp. reflexive-transitive) closure of this relation.

We also write begin(po_s) and end(po_s) to denote respectively the sets of the first and last abstract events of po_s. That is, if we imagine the po_s relation as a directed graph, then begin(po_s) contains the abstract events in po_s that do not have incoming edges, and end(po_s) contains the abstract events that do not have outgoing edges.

Competing pairs. The external communications coe $\cup$ rfe $\cup$ fre are over-approximated by the competing pairs relation cmp. In Figure 10(a), the cmp edges (a, f), (b_1, d), and (c, f) abstract in particular the fre edges (f', c') and (f', a'), and the rfe edge (b'_1, d') in Figure 10(b). We do not need to represent internal communications as they are already covered by po_s^+.

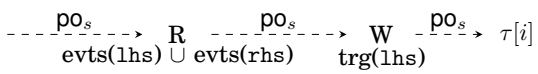
The cmp construction is similar to the first steps of static data race detection (see, e.g., [Kahlon et al. 2009, Sec. 5]), where statements involved in write-read or write-write communications are collected. As further work, we could reduce the set of competing

[3] We denote the function composition operator by $\circ$.

(1) assignment: $lhs = rhs; i$

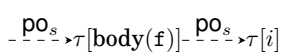
$\tau[lhs = rhs; i](aeg) =$

$$\begin{aligned}
&\textbf{let } \mathbb{E}_s' &&= aeg.\mathbb{E}_s \cup \mathbf{evts}(lhs) \cup \mathbf{evts}(rhs) \cup \\
&&&\quad \mathbf{trg}(lhs) \textbf{ in} \\
&\textbf{let } \mathsf{po}_s{}' &&= aeg.\mathsf{po}_s \cup \\
&&&\quad \mathbf{end}(aeg.\mathsf{po}_s) \times (\mathbf{evts}(rhs) \cup \mathbf{evts}(lhs)) \cup \\
&&&\quad (\mathbf{evts}(rhs) \cup \mathbf{evts}(lhs)) \times \mathbf{trg}(lhs) \\
&\textbf{in} \\
&\tau[i](\mathbb{E}_s', \mathsf{po}_s{}', aeg.\,\mathsf{cmp})
\end{aligned}$$



(2) function call[3]: $fun(); i$

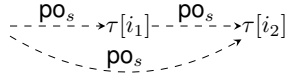
$\tau[fun(); i] = \tau[i] \circ \tau[\mathbf{body}(fun)]$



(3) guard: $[guard]\ i_1; i_2$

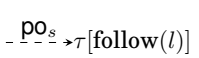
$\tau[\,[guard]\ i_1; i_2](aeg) =$

$$\begin{aligned}
&\textbf{let } \text{guarded} &&= \tau[i_1](aeg) \textbf{ in} \\
&\textbf{let } \mathbb{E}_s' &&= aeg.\mathbb{E}_s \cup \text{guarded}.\mathbb{E}_s \textbf{ in} \\
&\textbf{let } \mathsf{po}_s{}' &&= aeg.\mathsf{po}_s \cup \text{guarded}.\mathsf{po}_s \textbf{ in} \\
&\tau[i_2](\mathbb{E}_s', \mathsf{po}_s{}', aeg.\,\mathsf{cmp})
\end{aligned}$$



(4) unconditional forward jump: goto $l; i$

$\tau[goto\ l; i] = \tau[\mathbf{follow}(l)]$



(5) conditional backward jump: l: i_1; [$cond$] goto l; i_2

$\tau[l : i_1; [cond]\ \text{goto}\ l; i_2](aeg) =$

$$\begin{aligned}
&\textbf{let } \text{local} &&= \tau[i_1](aeg) \textbf{ in} \\
&\textbf{let } \mathbb{E}_s' &&= aeg.\mathbb{E}_s \cup \text{local}.\mathbb{E}_s \textbf{ in} \\
&\textbf{let } \mathsf{po}_s{}' &&= aeg.\mathsf{po}_s \cup \text{local}.\mathsf{po}_s \cup \mathbf{end}(\text{local}.\mathsf{po}_s) \\
&&&\quad \times \mathbf{begin}(\text{local}.\mathsf{po}_s) \textbf{ in} \\
&\tau[i_2](\mathbb{E}_s', \mathsf{po}_s{}', aeg.\,\mathsf{cmp})
\end{aligned}$$

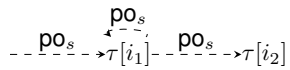


Fig. 11. Operations to create the aeg of a goto-program.

(6) assume / assert / skip: $\{\text{assume}, \text{assert}\}(\phi)$; i / skip; i

$\tau[\,assume(\phi); i] = \tau[i]$
$\tau[\,assert(\phi); i] = \tau[i]$
$\tau[\,skip; i] = \tau[i]$

$\xrightarrow{\mathsf{po}_s}\tau[i]$

---

(7) atomic section: atomic_begin; i_1; atomic_end; i_2

$\tau[atomic\_start; i_1; atomic\_end; i_2](aeg) =$

$$\begin{array}{l}\textbf{let}\ \text{section} = \tau[i_1]((aeg.\mathbb{E}_s \cup \{\mathsf{f}\},\\ \qquad\qquad aeg.\mathsf{po}_s \cup \text{end}(aeg.\mathsf{po}_s) \times \{\mathsf{f}\},\\ \qquad\qquad aeg.\,\mathsf{cmp}))\ \textbf{in}\\ \textbf{let}\ \mathsf{po}_s{}' = \ \text{section}.\mathsf{po}_s \cup \text{end}(\text{section}.\mathsf{po}_s) \times \{\mathsf{f}\}\ \textbf{in}\\ \tau[i_2]((\text{section}.\mathbb{E}_s, \mathsf{po}_s{}', \text{section}.\,\mathsf{cmp}))\end{array}$$

$\xrightarrow{\mathsf{po}_s}\mathsf{f}\xrightarrow{\mathsf{po}_s}\tau[i_1]\xrightarrow{\mathsf{po}_s}\mathsf{f}\xrightarrow{\mathsf{po}_s}\tau[i_2]$

---

(8) new thread: start_thread th; i

$\tau[start\_thread\ th; i](aeg) =$

$$\begin{array}{l}\textbf{let}\ \text{local} = \tau[\text{body}(th)](\bar{\emptyset})\ \textbf{in}\\ \textbf{let}\ \text{main} = \tau[i](aeg)\ \textbf{in}\\ \textbf{let}\ \text{inter} = \tau[i](\bar{\emptyset})\ \textbf{in}\\ (\text{local}.\mathbb{E}_s \cup \text{main}.\mathbb{E}_s,\ \ \text{local}.\mathsf{po}_s \cup \text{main}.\mathsf{po}_s,\ \ \text{local}.\mathbb{E}_s \otimes \text{inter}.\mathbb{E}_s)\end{array}$$

$\xrightarrow{\mathsf{po}_s}\tau[i]\xrightarrow{\mathsf{cmp}}\tau[\text{body}(f)]$

---

(9) end of thread: end_thread;

$\tau[end\_thread](aeg) = aeg$

$\emptyset$

Fig. 12. Operations to create the aeg of a goto-program (continued).

pairs using a synchronisation analysis, as in, e.g., [Sura et al. 2005]. If we assume the correctness of locks for example, some threads might never interfere.

Fences. In the aeg, we encode memory fences as special abstract events – i.e., as nodes in the graph. We write f for a full fence (e.g., mfence in x86, sync in Power, dmb in ARM), lwf for a lightweight fence (e.g., lwsync in Power), cf for a control fence (e.g., isync in Power, isb in ARM).

In [Alglave et al. 2010], fences are modelled as a relation *fenced* between concrete events. We did not use this approach, since a fence can then correspond to several edges and we would need an additional relation over our abstract event relations to keep

```
                                    mov r, #x
int local1=x;                       ldr r1,[r, #0]
local1=local1^local1;               exors r1, r1, r1
int local2=*(&y + local1);          mov r2, #y
                                    mov r3 [r2, r1]
```

Fig. 13. A fragment of C program (left) with two shared variables x and y, and a possible straightforward translation in ARM assembly (right).

track of the placement of fences. The effect of fences is interpreted during the cycle search in the aeg.

Dependencies form a relation between abstract events that is maintained outside of the aeg. They are calculated in musketeer from the input program on the C level.[4] Dependencies relate two accesses to shared memory at the assembly level via registers. As we analyse C programs and we make no assumption regarding the machine or the compiler used, neither the assembly translation nor the use of registers between shared memory accesses is unique or provided to us. For example, there is a dependency between x and y in the program fragment presented in Figure 13 (left), since a straightforward translation to assembly could generate a dependency. For example, in ARM assembly, we can translate this C program to the assembly code in Figure 13 (right), where a dependency by address between x and y was intentionally placed via the register r1. The register r1 always holds 0 after the interpretation of the exclusive disjunction on the value of x, and this 0 is added to the pointer to y before being dereferenced in register r3. Processors ignore that the value is always 0 and enforce a dependency.

Compiler optimisations can, however, remove these dependencies. In the tool musketeer, we provide the option *–no-dependencies* which safely ignores all these calculated dependencies in the aeg. Under this option, dependencies or fences might be spuriously inserted in places where an actual dependency already exists.

The same consideration applies to dependency generation, as we will treat in detail in Subsection 7.1.

5.4. Constructing Abstract Event Graphs from C Programs

We define a semantics of goto-programs in terms of abstract events, static program order and competing pairs. We give this semantics below by means of a case split on the type of the goto-instructions. Each of these cases is accompanied in Figures 11 and 12 by a graphical representation summarising the aeg construction on the right-hand side, and a formal definition of the semantics on the left-hand side. We assume that forward jumps are unconditional and that backward jumps are conditional. Conditional forward jumps can be "simulated" by those two instructions.

The construction of the aeg from a goto-program is implemented by means of a case split over the type of goto-instruction. The algorithm is outlined in Figure 15 (left side). We give further details about the algorithmin Section 5.7.

In Figures 11 and 12, we write $\tau[i]$ and $\tau[i_1; \ldots; i_n]$ to represent the semantics of a goto-instruction i and a sequence of goto-instructions $i_1; \ldots; i_n$, respectively. Other notations, e.g., follow(f) or body(f), are explained below. We do not compute the values of the variables, and thus do not interpret the expressions. In Figure 10(a), (a)Wx represents the assignment "x = input" in thread 1 in Figure 9 (since "input" is a local variable). This abstracts the values that "input" could hold, e.g., 1 (see (a')Wx1 in Figure 10(b))

[4]We keep track of the relations between local and shared variables per thread to calculate a dependency relation.

or 2 (see (a'')Wx2 in Figure 10(c)). Prior to building the aeg, we copy expressions of conditions or function arguments into local variables. Thus, communication via shared variables can occur in the assignment case only.

We now present the construction of the aeg starting with the intra-thread instructions (e.g., assignments, function calls), creating static events and po_s edges, and then the thread constructor, creating cmp edges. We write $G = \mathsf{aeg}(P) = \tau[P](\bar{\emptyset})$ for the aeg G corresponding to program P. The program P is a sequence of goto-instructions, and $\bar{\emptyset}$ denotes the empty aeg.

Assignments `lhs=rhs`. We decompose this statement into sets of abstract events: the reads from potential shared variables in `rhs` and `lhs`, denoted by evts(`rhs`) and evts(`lhs`), and the writes to the potential target objects trg(`lhs`). We do not assume any order in the evaluation of the variables in an expression. Hence, we connect to the incoming po_s all the reads of `rhs` and all the reads of `lhs` except trg(`lhs`). We then connect each of them to the potential target writes trg(`lhs`). The functions evts() and trg() return sets of abstract events with address specifiers according to the pointer analysis (cf. Section 5.2).

We also record the data and address dependencies between abstract events. For instance, if we have `int r1=x; int r2=r1; *(&y+r2)=1` we know that the abstract event Rx from the rhs of the first instruction is in dependency with the write to `r2` (via the use of `r1`). Moreover, the Wa issued by the last statement (with the address specifier `a` corresponding to the expression `*(&y+r2)`) depends on `r2`, and thus there is a dependency between Rx and Wa.

Function calls fun(). We build the po_s corresponding to the function's body (written body(fun)). We then replace the call to a function fun() by its body. This ensures a better precision, in the sense that a function can be fenced in a given context and unfenced in another. In our tool musketeer, recursive functions must be inlined up to a bound. musketeer issues warnings when encountering recursive functions.

Guarded statement. We do not keep track of the values of the variables at each program point. Thus we cannot evaluate the guard of a statement. Hence, we abstract the guard and make this statement non-deterministically reachable, by adding a second po_s edge, bypassing the statement.

Unconditional forward jump to a label L. We connect the previous abstract events to the next abstract events that we generate from the program point L. In Figure 11, we write follow(L) for the sequence of statements following the label L.

Conditional backward jump. A conditional backward jump is a jump to a label already visited. It arises from loops in the program. We copy and append the po_s subgraph between the label and the goto twice, and connect the last abstract event of the copy to the first abstract event of the original body with a po_s edge. In Figure 11, begin(S) and end(S) are respectively the sets of the first and last abstract events of the po_s sub-graph S. We assume that the statement denoted by i_1 (i.e., the body of the loop) does not contain a jump statement that jumps outside of the body.

Assumption, assertion, skip. Similarly to the guarded statement, as we cannot evaluate the condition, we abstract the assumptions and assertions by bypassing them. They are thus handled the same as the skip statement.

Atomic sections. The atomic sections in a goto-program model idealised atomic sections without having to rely on the correctness of their implementation. In the CProver framework, we use the constructs `__CPROVER_atomic_begin` and `__CPROVER_atomic_end`. Atomic sections are used in many theoretical concurrency and verification works. For example, we use them (see Section 7) for copying data to atomic structures, as in,

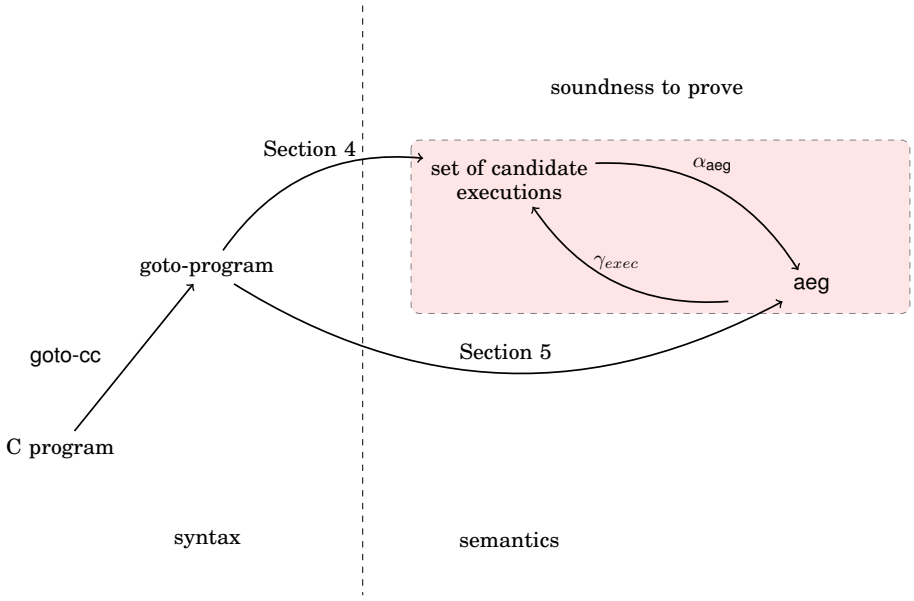


Fig. 14. From C programs to aegs and candidate executions

e.g., our implementation of the Chase-Lev queue [Chase and Lev 2005] or for implementing compare-and-swaps, as in, e.g., our implementation of Michael and Scott's queue [Michael and Scott 1996].

In this work we overapproximate atomic sections by only considering the effect on memory ordering of entering and leaving a critical section. We do not model the atomicity aspect of atomic sections. Also handling this aspect could improve performance and precision as less (spurious) interferences have to be considered.

We model atomic sections by placing two full fences, written f in Figure 12, right after the beginning of the section and just before the end of the section.

Construction of cmp. We also compute the competing pairs that abstract external communications between threads. For each abstract event with address specifier s, we augment the cmp relation with pairs made of this abstract event and abstract events from an interfering thread with a compatible address specifier s' (i.e., the sets of memory locations represented by s and s' overlap). One of the two accesses needs to be a write. These cmp edges abstract the relations coe, fre, and rfe.

In Figure 12, we use $\otimes$ to construct the cmp edges when a new interfering thread is spawned. That is, when the goto-instruction start_thread is met. We define $\otimes$ as $A \otimes B = \{(a,b) \in A \times B \mid \mathsf{comp}(\mathsf{addr}(a), \mathsf{addr}(b)) \wedge (\mathsf{write}(a) \vee \mathsf{write}(b))\}$. We write $\bar{\emptyset}$ for the triple $(\emptyset, \emptyset, \emptyset)$ representing the empty aeg.

5.5. Correctness of the aeg construction

In this section, we explain why the aeg constructed with the transformers of Figures 11 and 12 captures all the candidate executions of a program. As a consequence, if we search for cycles in the aeg we can find all corresponding cycles that might occur in a concrete execution.

Given a goto-program, we want to show that for any event structure that could be derived from it and for any candidate execution valid for this event structure, the sets

of events ($\mathbb{E}$) and all the relations (po, rf, fr, co) are contained (in a sense defined in the following paragraph) in their static counterparts $\mathbb{E}_s$, po_s^+, and cmp. We need here the transitive closure of po_s as the po relation of a concrete execution is transitive, but the static program order relation po_s of an aeg is not transitive.

We refer in the following to $\mathbb{E}_s$ as the set of static events, and to $\mathbb{E}$ as the set of dynamic events. Similarly, we refer to po_s as the static program order, and to po as the dynamic program order. The reason why we choose the terms static and dynamic is that $\mathbb{E}_s$, po_s, and cmp are computed by our static analysis, whereas $\mathbb{E}$, po, rf, fr, and co refer to the events and relations that represent a concrete candidate execution. We further use the symbols se and $srel$ to refer to a set of static events and a static relation, respectively.

To map the static events to dynamic events, and relations over static events to relations over dynamic events, we define $\gamma : \wp(\mathbb{E}_s \times \mathbb{E}_s) \to \wp(\mathbb{E} \times \mathbb{E})$ which *concretises* a static relation as the union of all the dynamic relations that it could correspond to, and $\gamma_e : \wp(\mathbb{E}_s) \to \wp(\mathbb{E})$ which concretises a set of static events as the union of all the sets of dynamic events that they could correspond to.[5] These two functions are a formalisation of how we interpret aegs.

More formally, we define $\gamma_e(se) \triangleq \{e' \mid \exists e \in se \text{ s.t. } \mathrm{comp}(\mathrm{addr}(e), \mathrm{addr}(e')) \wedge \mathrm{dir}(e) = \mathrm{dir}(e') \wedge \mathrm{origin}(e) = \mathrm{origin}(e')\}$, where $\mathrm{origin}(a)$ returns the syntactical object in the source code from which either a concrete or abstract event was extracted.[6] In this definition e' is a concrete event whereas e is a static event. The function $\mathrm{comp}(\mathrm{addr}(e), \mathrm{addr}(e'))$ indicates whether the memory address accessed by e' and the address specifier of e are compatible (i.e., whether the result $\mathrm{addr}(e)$ computed by the pointer analysis includes the memory address $\mathrm{addr}(e')$; see Section 5.2 for details). The function $\mathrm{dir}()$ (for direction) indicates whether the static event or event given is a read or write. Thus, $\mathrm{dir}(e) = \mathrm{dir}(e')$ requires that the static event and the concrete event are either both reads or both writes. We use γ_e to define γ: $\gamma(srel) \triangleq \{(c_1, c_2) \mid \exists (s_1, s_2) \in srel \text{ s.t. } (c_1, c_2) \in \gamma_e(\{s_1\}) \times \gamma_e(\{s_2\})\}$. In this definition c_1, c_2 are concrete events and s_1, s_2 are static events. We next show a lemma about γ_e and γ that we will use later on.

LEMMA 5.1. *Let* $\mathbb{E}_1 \subseteq \gamma_e(\mathbb{E}_{s,1})$ *and* $\mathbb{E}_2 \subseteq \gamma_e(\mathbb{E}_{s,2})$. *Then* $\mathbb{E}_1 \times \mathbb{E}_2 \subseteq \gamma(\mathbb{E}_{s,1} \times \mathbb{E}_{s,2})$.

PROOF. Let $(c_1, c_2) \in \mathbb{E}_1 \times \mathbb{E}_2$. Then $c_1 \in \gamma_e(\mathbb{E}_{s,1})$ and $c_2 \in \gamma_e(\mathbb{E}_{s,2})$. Using the definition of γ_e, there are s_1, s_2 such that $c_1 \in \gamma_e(\{s1\})$ and $c_2 \in \gamma_e(\{s_2\})$. Then, since $(s_1, s_2) \in \mathbb{E}_{s,1} \times \mathbb{E}_{s,2}$, it follows by the definition of γ that $(c_1, c_2) \in \gamma(\mathbb{E}_{s,1} \times \mathbb{E}_{s,2})$. □

Figure 14 shows the relationship between goto-programs, candidate executions, and aegs. In our proof, we show that for a given program, any execution of that program has its events and relations contained in the γ_e, γ of the corresponding aeg.

To go from goto-programs to a set of event structures (and then a set of candidate executions), we describe formally how the event structures can be generated. This is a formalisation of the informal description of how event structures are generated from a program as described in Section 4.

For the definition of the assignment case we make use of the function dyn_evts(`lhs = rhs`). It takes an assignment statement, and returns all possible events it might correspond to in a concrete execution. For example, for `x=*p+z`, we have

$$\text{dyn\_evts}(\texttt{x=*p+z}) = \bigcup_{i=1,\ldots,n} \{(Wxv_1, \{Ry_iv_2, Rzv_3\}) \mid v_1 = v_2 + v_3\}$$

[5] We usually use γ_e with a singleton set (containing one static event) as an argument.
[6] The function origin is dual to evts, which returns events given an expression or instruction.

That is, dyn_evts() yields a set of pairs with the first component being the write event, and the second component being the set of read events. Moreover, $y_1, \ldots, y_n$ are the concrete memory locations the pointer p might point to according to the pointer analysis. We thus presuppose a correct pointer analysis (cf. Section 5.2).

We next formalise the generation of event structures from a given program P (cf. Section 4) by means of the function σ (which is analogous to τ for the aeg construction). We write $S(P) = \sigma(P)(\emptyset)$ for the set of event structures of a program P, with $\emptyset$ denoting the empty set of event structures.

Definition 5.2 (*event structure semantics S for goto-programs*). We write ses for a set of event structures. We define the semantics as follows:

(1) *assignment*:
$\sigma[lhs = rhs; i](\mathsf{ses}) =$
let $de = \mathrm{dyn\_evts}(lhs = rhs)$ **in**
let $\mathsf{ses}' = \bigcup_{es \in \mathsf{ses}} (\bigcup_{(t,r) \in de} \{(es.\mathbb{E} \cup \{t\} \cup r, es.\mathsf{po} \cup \mathsf{end}(es.\mathsf{po}) \times r \cup r \times \{t\}))$ **in**
$\sigma[i](\mathsf{ses}')$

(2) *function call*:
$\sigma[fun(); i] = \sigma[i] \circ \sigma[\mathrm{body}(fun)]$

(3) *guarded statement*:
$\sigma[\,[guard]i_1; i_2](\mathsf{ses}) =$
let $\mathsf{ses}' = \sigma[i_1](\mathsf{ses}) \cup \mathsf{ses}$ **in**
$\sigma[i_2](\mathsf{ses}')$

(4) *unconditional forward jump*:
$\sigma[goto\ l; i] = \sigma[\mathrm{follow}(l)]$

(5) *conditional backward jump*[7] :
$\sigma[l : i_1; [cond]\mathsf{goto}\ l; i_2](\mathsf{ses}) =$
$\bigcup_{n \in \mathbb{N}} \sigma[i_2](\sigma[i_1]^n(\mathsf{ses}))$

(6) *assume / assert / skip*:
$\sigma[assume(\phi); i] = \sigma[i]$

(7) *atomic section*:
$\sigma[atomic\_start; i_1; atomic\_end; i_2](\mathsf{ses}) =$
let section = $\sigma[i_1](\bigcup_{es \in \mathsf{ses}} \{(es.\mathbb{E} \cup \{\mathsf{f}\}, es.\mathsf{po} \cup \mathsf{end}(es.\mathsf{po}) \times \{\mathsf{f}\})\})$ **in**
$\sigma[i_2](\bigcup_{es \in \mathrm{section}} \{(es.\mathbb{E}, es.\mathsf{po} \cup \mathsf{end}(es.\mathsf{po}) \times \{\mathsf{f}\})\})$

(8) *new thread*:
$\sigma[start\_thread\ th; i](\mathsf{ses}) =$
let local = $\sigma[\mathrm{body}(th)](\emptyset)$ **in**
let main = $\sigma[i](\mathsf{ses})$ **in**

[7] We use $f^n(x)$ to denote $\underbrace{f(\ldots f}_{n \text{ times}}(x))$.

$\bigcup_{es_1 \in \text{local}, es_2 \in \text{main}} \{(es_1.\mathbb{E} \cup es_2.\mathbb{E},\ es_1.\mathsf{po} \cup es_2.\mathsf{po})\}$

(9) *end of thread*:
$\sigma[end\_thread](\mathsf{ses}) =$
$\bigcup_{es \in ses} \{(es.\mathbb{E},\ es.\mathsf{po}^+)\}$

Similarly as for aegs, begin(po) and end(po), respectively, denote the sets of events that have no incoming po edges or no outgoing po edges.

The guards are not interpreted by this semantics, meaning that there may be event structures that do not correspond to an actual execution of the program, regardless of the architecture. For example, even if an if-condition could never be satisfied, an event structure passing through the body of this if-statement would be constructed. These kinds of event structures are thus spurious. However, since this is an overapproximation we still arrive at a correct soundness argument if we can show that all those event structures are contained in the aeg.

Using this formalisation, we can establish that, for each event structure of a goto-program, we can find po edges and events, abstracted by po_s^+ edges and abstract events in the aeg (Lemma 5.3). This can be proven by structural induction over the event structure transformers of Definition 5.2 and the corresponding aeg transformers of Figures 11 and 12. We then need to show that for each candidate execution that corresponds to an event structure of the goto-program, all the communication relations (rf, co, and fr) are captured by the cmp relation in the aeg (Lemma 5.4). More precisely, we want to establish the three properties below:

PROPERTY 1 (SOUNDNESS). *Let P be a program, let $E \in \boldsymbol{S}(P)$ and let $X = (\mathsf{rf}, \mathsf{co}, \mathsf{fr})$ be an execution witness. Let further $G = \mathsf{aeg}(P)$. Then the aeg G soundly captures (E, X) if the following properties hold:*

$$\gamma(G.\mathsf{po}_s^+) \supseteq E.\mathsf{po} \cup X.\mathsf{coi} \cup X.\mathsf{rfi} \cup X.\mathsf{fri}, \tag{1}$$
$$\gamma(G.\,\mathsf{cmp}) \supseteq X.\mathsf{coe} \cup X.\mathsf{rfe} \cup X.\mathsf{fre}, \tag{2}$$
$$\gamma_e(G.\mathbb{E}_s) \supseteq E.\mathbb{E}, \tag{3}$$

The property above corresponds to γ_{exec} from Figure 14. That is, γ_{exec} works by concretising the static relations $G.\mathsf{po}_s^+$ and $G.\,\mathsf{cmp}$ via γ, and the set of static events $G.\mathbb{E}_s$ via γ_e.

LEMMA 5.3 (EVENTS AND PO). *For any event structure $E \in S(P)$, and aeg $G = \mathsf{aeg}(P)$ we have $E.\mathbb{E} \subseteq \gamma_e(G.\mathbb{E}_s)$ and $E.\mathsf{po} \subseteq \gamma(G.\mathsf{po}_s^+)$.*

PROOF. The proof works by structural induction over the input program P, and thus by considering each event structure transformer in Definition 5.2, and the corresponding aeg transformer in Figures 11 and 12. We use the notation $\mathsf{ses} \ll G \triangleq \forall E \in \mathsf{ses}\colon E.\mathbb{E} \subseteq \gamma_e(G.\mathbb{E}_s) \wedge E.\mathsf{po} \subseteq \gamma(G.\mathsf{po}_s^+)$ in the following. We further define $Prop(P) \triangleq \forall \mathsf{ses}, G\colon \mathsf{ses} \ll G \Rightarrow \sigma[P](\mathsf{ses}) \ll \tau[P](G)$. We thus prove in the following that for all input programs P the predicate $Prop(P)$ holds.

The base case of the induction is the "end of thread" case. Assume $\mathsf{ses} \ll G$. The only change the concrete transformer for the end_thread case makes to ses is making the po relation of the contained event structures transitive. The aeg transformer for end_thread does not change the aeg G. Since in the definition of $\ll$ the transitive closure of the static program order relation of the aeg is taken it follows that $\sigma[end\_thread](\mathsf{ses}) \ll \tau[end\_thread](G)$ holds.

In the induction step for each case, we can assume the predicate $Prop$ for the constituents of the respective code fragment. For example, in the assignment case $lhs := rhs; i$ (cf. Figure 11 and Definition 5.2) we can assume $Prop(i)$.

Exemplary for all the cases, we show below the proof steps for the assignment case and the conditional backward jump case (which is used to implement loops).

Assignment transformer. We show $Prop(lhs := rhs; i)$. Let $Prop(i)$ hold. Let ses be a set of event structures and let G be an aeg such that ses $\ll G$. Let $es \in$ ses, let $st =$ "`lhs = rhs`", and let $(t, r) \in$ dyn_evts(st).

We first show the set inclusion for the set of events and the set of concretised static events. We use the abbreviation evts $=$ evts$(lhs) \cup$ evts(rhs) in the following. We have to show that $es'.\mathbb{E} = es.\mathbb{E} \cup \{t\} \cup r \subseteq \gamma_e(G.\mathbb{E}_s \cup$ evts $\cup$ trg$(lhs)) = \gamma_e(G'.\mathbb{E})$. By the definition of γ_e we get $\gamma_e(G'.\mathbb{E}) = \gamma_e(G.\mathbb{E}_s) \cup \gamma_e($evts$) \cup \gamma_e(trg(lhs))$. By the assumption of ses $\ll G$ we have $es.\mathbb{E} \subseteq \gamma_e(G.\mathbb{E}_s)$. By the correctness of the pointer analysis, we further have $r \subseteq \gamma_e($evts$)$ and $\{t\} \subseteq \gamma_e(trg(lhs))$. We thus have $es'.\mathbb{E} \subseteq \gamma_e(G'.\mathbb{E})$.

We next show the set inclusion for the program order relation and the concretised static program order relation. We have to show that $es'.$po $= es.$po $\cup$ end$(es.$po$) \times r \cup r \times \{t\} \subseteq \gamma(G.po_s \cup$ end$(G.$po$_s) \times$ evts $\cup$ evts $\times$ trg$(lhs)) = \gamma(G'.$po$_s)$. By the definition of γ we have $\gamma(G'.$po$_s) = \gamma(G.$po$_s) \cup \gamma($end$(G.$po$_s) \times$ evts$) \cup \gamma($evts $\times$ trg$(lhs))$. By the assumption of ses $\ll G$ we have $es.$po $\subseteq \gamma(G.$po$_s)$. Thus, we also have end$(es.$po$) \subseteq \gamma_e($end$(G.$po$_s))$. By Lemma 5.1 we thus get end$(es.$po$) \times r \subseteq \gamma(end(G.po_s) \times$ evts$)$ and $r \times \{t\} \subseteq \gamma($evts $\times$ trg$(lhs))$. We thus have $es'.$po $\subseteq \gamma(G'.$po$_s)$.

Let ses$'$ denote the set of event structures resulting from applying the assignment transformer to all event structures in ses. Since we have above chosen $es \in$ ses arbitrarily we have ses$' \ll G'$. Then by assumption of $Prop(i)$ we get $\sigma[i]($ses$') \ll \tau[i](G')$.

Conditional backward jump transformer. The concrete transformer for the conditional backward jump case (Definition 5.2) involves an infinite union over the number of times the loop body is executed. We show that for any number of iterations n, the resulting event structures are contained in the concretisation of the corresponding aeg.

Let $Prop(i_1)$ and $Prop(i_2)$ hold. Let further ses be a set of event structures and let G be an aeg such that ses $\ll G$. Now let n be an arbitrary positive integer. Then since $Prop(i_1)$ and $Prop(i_2)$ hold we get get $\sigma[i_2](\sigma[i_1]^n(ses)) \ll \tau[i_2](\sigma[i_1]^n(G)))$. The term $\tau[i_2](\tau[i_1]^n(G))$ is similar to the aeg interpretation of $P' = i_2; i_1, \ldots, i_n$. The concretisation of P' is contained in the concretisation of $P'' = l : i_1[cond]goto\ l; i_2$ due to the back-edge introduced in the aeg construction. Therefore, we also have $\sigma[i_2](\sigma[i_1]^n(ses)) \ll \tau[P''](G))$. Since we have chosen n arbitrarily it follows that $\bigcup_{n\in\mathbb{N}} \sigma[i_2](\sigma[i_1]^n(ses)) \ll \tau[P''](G)$. □

LEMMA 5.4 (RF, CO AND FR). *For any event structure $E \in S(P)$ with candidate execution (E, X) (with X = (rf, co, fr)) and $G =$ aeg(P) we have* rfe, coe, fre $\subseteq \gamma(G.$ cmp*).*

PROOF. We first show that $X.$rfe (the external read-from edges, cf. Section 4) is contained in $\gamma(G.$ cmp$)$. For any pair $(a, b) \in X.$rfe, there is a pair of abstract events c, d in the aeg that abstracts these events (by Lemma 5.3). Since a, b are in the rfe relation, they access the same memory location. Then, by the correctness of the pointer analysis, and the construction of $\otimes$ in Figure 12, the events c, d have compatible address specifiers (i.e., comp(addr(c), addr(d))), and occur on different threads. The two events are thus connected by a cmp edge. Thus, $(a, b) \in \gamma(\{(c, d), (d, c)\})$. Therefore, rfe $\subseteq \gamma(G.$ cmp$)$, as γ is monotonic. With the same argument we can also show coe, fre $\subseteq \gamma(G.$ cmp$)$. □

We can now conclude that any execution is contained in the aeg.

THEOREM 5.5 (ANY EXECUTION IS CONTAINED IN THE AEG). *Let P be a program, let (E, X) be a candidate execution of P, and let $G = \mathsf{aeg}(P)$. Then Property 1 holds for (E, X) and G.*

PROOF. We combine Lemma 5.3 and Lemma 5.4. □

We next show that any critical cycle in the framework of Alglave et al. [2010] is also present as a cycle in the aeg.

THEOREM 5.6 (STATIC CRITICAL CYCLES). *Let P be a program. Let (E, X) be a candidate execution of P containing a critical cycle $c = c_0, \ldots, c_{n-1}$ (with the c_i denoting events connected by po, rf, co, or fr edges) with respect to some architecture A. Then there is a corresponding cycle $d = d_0, \ldots, d_{n-1}$ in $G = \mathsf{aeg}(P)$ such that (1) $c_0, \ldots, c_{n-1} \in \gamma_e(\{d_0, \ldots, d_{n-1}\})$ and (2) $\{(c_i, c_{i+1 \bmod n}) \mid 0 \leq i < n\} \subseteq \gamma(\{(d_i, d_{i+1 \bmod n}) \mid 0 \leq i < n\})$.*

PROOF. By Lemma 5.3, and the definition of γ_e, for each c_i there is a uniquely defined abstract event d_i such that $c_i \in \gamma_e(\{d_i\})$. This implies (1). Since c_i and $c_{i+1 \bmod n}$ are connected by an edge, by Lemma 5.4, and the definition of γ, there must be an edge between the corresponding abstract events d_i and $d_{i+1 \bmod n}$. Hence, the sequence of abstract events $d_0, \ldots, d_{n-1}$ forms a cycle in the aeg. Then, by the definition of γ, we get (2). □

By the above theorem, we can find all the critical cycles of all candidate executions by collecting the corresponding critical cycles in the aeg. We next explain a special case in the aeg cycle search that appears due to the presence of loops in the analysed program.

5.6. Loops, arrays, and pointers

In the previous section we have shown that the aeg encompasses all candidate executions of a program. Thus, via a cycle search in the aeg we can find all the corresponding critical cycles in all the candidate executions. However, this might require visiting some of the aeg nodes twice. Specifically, this may be necessary for cycles that contain two memory accesses from the same thread that originate from the same instruction which occurs in a loop. To be able to still use a common graph cycle search algorithm that only finds simple cycles, we duplicate the loop body (for certain loops) in the aeg before doing the cycle search. Below we give an account of our approach examining all the possible cases that might occur.

In goto-programs, loops from the original C program appear as goto statements that jump back to earlier instructions. If we build our aeg directly following the goto-program, with a po_s back-edge connecting the end of the loop body to its entry, we already can find most of the critical cycles. Recall from Section 4 that in a critical cycle there are at most two events per thread, and the events target different locations. Let us analyse the cases. The first case is an iteration i of this loop on which a critical cycle connects two events (a_i) and (b_i) that both originate from different instructions in the loop body. The critical cycle will be captured by its static counterpart that abstracts in particular these two events with some abstract events (a) and (b).

Now, for a given execution, if a critical cycle connects the event (a_i) of an iteration i to the event (b_j) (originating from a different instruction) of a later iteration j (i.e., $i < j$), then these events are abstracted respectively by some abstract events (a) and (b) in the aeg. As we do not evaluate the expressions, we abstracted the loop guard and any local variable that would vary across the iterations. Thus, all the iterations can be statically captured by one abstract representation of the body of the loop. Then, thanks to the po_s back-edge, any critical cycle involving (a_i) and (b_j) is abstracted by a static critical cycle relating (a) and (b), even though (b) might be before (a) in the body of the loop.

The only case that is not handled by this approach is when (a_i) and (b_j) originate from the same instruction and are abstracted by the same abstract event (c). As the variables addressed by the events on the same thread of a critical cycle need to be different, this case can only occur when (a_i) and (b_j) are accessing an array or a pointer whose index or offset depends on the iteration. We do not evaluate these offsets or indices, which implies that two accesses to two distinct array positions might be abstracted by the same abstract event (c).

In order to detect such critical cycles with a simple cycle search, we copy the body of the loop (not including the po_s back-edge for the copy). Hence, a static critical cycle will connect (c) in the first instance of the body and (c') in the second instance of the body to abstract the critical cycle involving (a_i) and (b_j). We detail the loop duplication algorithm in the next section.

5.7. Loop duplication algorithm

The aeg construction algorithm is given in pseudo-code in Figure 15. The loop duplication is handled in the "backward jump" case of the aeg construction. The aeg construction is invoked by calling construct_aeg(ins, $\emptyset$), with the first argument being the first instruction in the goto-program (i.e., the entry point), and the second argument being an empty set of aeg nodes. The function builds the aeg by adding nodes (i.e., abstract events) and edges (static program order po_s or competing pairs cmp) to a global, initially empty aeg.

The function construct_aeg() performs a case distinction over the type of instruction. After the current instruction has been handled, it calls construct_aeg() again to handle the next instruction (see, e.g., line 9). We assume that next(ins) returns the instruction that immediately follows ins, that tgt(ins) returns the jump target of the instruction ins (for jump instructions, see Figure 11), that function(ins) returns the function called by the instructions ins and that body(fun) returns the first instruction of the function fun.

We make use of three functions that return and may add a set of abstract events to the global aeg. The function get_abs_evts(ins) returns the abstract events corresponding to an instruction. These abstract events can be both memory reads and writes. If the abstract events have not been previously constructed, the function may query the result of the pointer analysis to find the potential memory locations the instruction might access. Then the abstract events are added to the aeg. The function get_abs_rds($expr$) returns and constructs the abstract read events corresponding to the given expression. The function get_abs_wrs($expr$) returns and constructs the abstract write events corresponding to the given expression.

The function add_pos_edges($evts_1$, $evts_2$) adds static program order edges to the aeg from all the abstract events in set $evts_1$ to all the abstract events in set $evts_2$. The function add_pos_edge(e_1, e_2) adds a static program order edge to the aeg between the abstract events e_1 and e_2.

We next detail the actual loop duplication which is handled in the "backward jump" case of the aeg construction. Our implementation duplicates the loop bodies only for loops that contain accesses to arrays or accesses via pointers for which the target location might depend on the loop iteration (line 18). The function contains_pointer_access(ins_1, ins_2) returns ***true*** when the section of the goto-program between instructions ins_1 and ins_2 contains accesses to arrays or pointers for which the target location might depend on the loop iteration.

The subroutine duplicate_body($start$, end) duplicates the loop body corresponding to a back-edge (corresponding to a backwards goto statement in the goto-program). It makes use of the function evts_between($start$, end) which returns the set of abstract events in the current, partially constructed aeg which are on a path from an abstract event in $start$ to an abstract event in end. It also uses the function get_pos_successors(e) which

```
1  function construct_aeg(ins, prev_evts)
2  switch type(ins) do
       /* Cases for all the transformers of
          Figures 11 and 12                    */
3      case assignment
4          evts_1 = get_abs_rds(lhs(ins))
5          evts_2 = get_abs_rds(rhs(ins))
6          evts_3 = get_abs_wrs(lhs(ins))
7          add_pos_edges(prev_evts, evts_1 ∪ evts_2)
8          add_pos_edges(evts_1 ∪ evts_2, evts_3)
9          construct_aeg(next(ins), evts_3)
10     endsw
11     case function call
12         construct_aeg(body(function(ins)),
           prev_evts)
13     endsw
14     ...
15     case backward jump
16         evts_tgt = get_abs_evts(tgt(ins))
17         add_pos_edges(prev_evts, nodes_tgt)
18         if contains_pointer_access(tgt(ins), ins)
           then
19             new_prev =
               duplicate_body(evts_tgt, prev_evts)
               construct_aeg(next(ins), new_prev)
20         else
21             construct_aeg(next(ins), prev_evts)
22         end
23     endsw
24 endsw
```

```
1  function duplicate_body(start, end)
2  old_to_new = {}
3  loop = evts_between(start, end)
4  for e ∈ loop do
5      new_e = copy(e)
6      old_to_new[e] = new_e
7  end
8  for e ∈ loop do
9      new_e = old_to_new[e]
10     succs = get_pos_successors(e)
11     for s ∈ succs do
12         new_s = old_to_new[s]
13         add_pos_edge(new_e, new_s)
14     end
15 end
16 new_start = {old_to_new[e] | e ∈ start}
17 add_pos_edges(end, new_start)
18 new_end = {old_to_new[e] | e ∈ end}
19 return new_end
```

Fig. 15. aeg construction algorithm (including loop duplication algorithm)

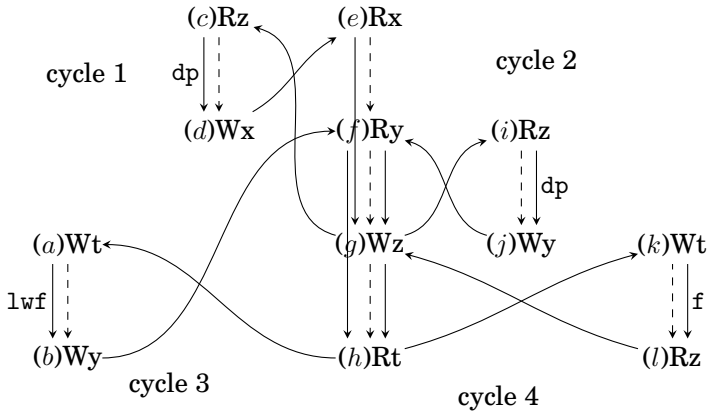
returns the set of abstract events to which there is a static program order edge in the current, partially constructed aeg from the abstract event e.

The function duplicate_body($start$, end) first copies all the abstract events in the loop body (lines 4–7). Then, it connects the new abstract events via static program order edges (lines 8–15). Finally, it attaches the new loop body to the old loop body (lines 16–17), and returns the abstract events at the end (line 19).

The loop duplication algorithm also works correctly for nested loops. Nested loops in the original C program appear in the goto-program as backward jumps from a location i to a location $j < i$, such that there is another backward jump from location $k > i$ to a location $l < j$ (representing the outer loop). The number of aeg nodes introduced by the loop duplication for a given loop grows quadratically with the nesting depth (i.e., the number of other loops it contains). A formal argument showing this can be found in [Nimal 2015].

5.8. Cycle detection

Once we have the aeg, we enumerate (using Tarjan's algorithm [Tarjan 1973]) its potential critical cycles by searching for cycles that contain at least one edge that corresponds to a delay, as defined in Section 4.



$$
\begin{array}{ll}
\textbf{min} & \mathtt{dp}_{(e,g)} + \mathtt{dp}_{(f,h)} + \mathtt{dp}_{(f,g)} + 3 \cdot (\mathtt{f}_{(e,f)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)}) \\
& +2 \cdot (\mathtt{lwf}_{(e,f)} + \mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)}) \\
\textbf{s.t.} & \text{cycle 1, delay } (e,g)\text{: } \mathtt{dp}_{(e,g)} + \mathtt{f}_{(e,f)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(e,f)} + \mathtt{lwf}_{(f,g)} \geq 1 \\
& \text{cycle 2, delay } (f,g)\text{: } \mathtt{dp}_{(f,g)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(f,g)} \geq 1 \\
& \text{cycle 3, delay } (f,h)\text{: } \mathtt{dp}_{(f,h)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)} + \mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)} \geq 1 \\
& \text{cycle 4, delay } (g,h)\text{: } \mathtt{f}_{(g,h)} \geq 1
\end{array}
$$

Fig. 16. Example of resolution with between.

6. SYNTHESISING FENCES

In this section, we describe how to compute which types of fences to place at which locations in the program, in order to forbid the weak behaviours embodied by the potential critical cycles contained in the aeg.

6.1. Options for placing fences

In Figure 16, we give an aeg with five threads: $\{a, b\}, \{c, d\}$, $\{e, f, g, h\}$, $\{i, j\}$ and $\{k, l\}$. Each node is an abstract event computed as described in the previous section. The dashed edges represent the po_s between abstract events in the same thread. The full lines represent the edges involved in a cycle. The aeg of Figure 16 has four potential critical cycles. We derive the set of constraints in a process we define later in this section. We now have a set of cycles to forbid by placing fences. Moreover, we want to optimise the placement of the fences.

Challenges. If there is only one type of fence (as in TSO, which only features mfence), optimising only consists of placing a minimal number of fences to forbid as many cycles as possible. For example, placing a full fence sync between f and g in Figure 16 might forbid cycles 1, 2 and 3 under Power, whereas placing it somewhere else might forbid at best two amongst them.

Since we handle several types of fences for a given architecture (e.g., dependencies, lwsync and sync on Power), we can also assign some cost to each of them. For example, following the folklore, a dependency is less costly than an lwsync, which is itself less costly than a sync. Given these costs, one might want to minimise their sum along different executions: to forbid cycles 1, 2 and 3 in Figure 16, a single lwsync between f and g can be cheaper at runtime than three dependencies respectively between e and g, f and g, and f and h. However, if we had only cycles 1 and 2, the dependencies would be cheaper. We see that we have to optimise both the placement and the type of fences at the same time.

Input: aeg ($\mathbb{E}_s$,po_s,cmp) and potential critical cycles $C = \{C_1, \ldots, C_n\}$
Problem: minimise $\sum_{(l,\mathsf{t})\in\text{potential-places}(C)} \mathsf{t}_l \times \mathsf{cost}(\mathsf{t})$
Constraints: for all $d \in \text{delays}(C)$
(* for TSO, PSO, RMO, Power *)
if $d \in \mathsf{poWR}$ then $\sum_{e\in\mathsf{between}(d)} \mathsf{f}_e \geq 1$
if $d \in \mathsf{poWW}$ then $\sum_{e\in\mathsf{between}(d)} \mathsf{f}_e + \mathsf{lwf}_e \geq 1$
if $d \in \mathsf{poRW}$ then $\mathsf{dp}_d + \sum_{e\in\mathsf{between}(d)} \mathsf{f}_e + \mathsf{lwf}_e \geq 1$
if $d \in \mathsf{poRR}$ then $\mathsf{dp}_d + \sum_{e\in\mathsf{between}(d)} \mathsf{f}_e + \mathsf{lwf}_e \quad + \sum_{e\in\mathsf{ctrl}(d)} \mathsf{cf}_e \geq 1$
(* for Power *)
if $d \in \mathsf{cmp}$ then $\sum_{e\in\mathsf{cumul}(d)} \mathsf{f}_e + \sum_{e\in\mathsf{cumul}(d)\cap\neg\mathsf{poWR}\cap\neg\mathsf{poRW}} \mathsf{lwf}_e \geq 1$
Output: the set $\mathsf{actual\text{-}places}(C)$ of pairs (l, t) s.t. t_l is set to 1 in the ILP solution

Fig. 17. ILP for inferring fence placements.

We model our problem as an *integer linear program* (ILP) (see Figure 17; Figure 16 gives an instantiation of the ILP for the aeg shown in the same figure). Solving our ILP gives us a set of fences to insert to forbid the cycles. This set of fences is optimal in that it minimises the cost function. More precisely, the constraints are the cycles to forbid, each variable represents a fence to insert, and the cost function sums the cost of all fences. We now explain how to construct this ILP.

6.2. Cost function of the ILP

We handle several types of fences: full (f), lightweight (lwf), control fences (cf), and dependencies (dp). On Power, the full fence is sync, the lightweight one lwsync. We write $\mathbb{T}$ for the set $\{\mathsf{dp}, \mathsf{f}, \mathsf{cf}, \mathsf{lwf}\}$. We assume that each type of fence has an *a priori* cost (e.g., a dependency is cheaper than a full fence), regardless of its location in the code.[8] We write $\mathsf{cost}(\mathsf{t})$ for $\mathsf{t} \in \mathbb{T}$ for this cost.

We take as input the aeg of our program and the potential critical cycles to fence. We define two sets of pairs (l, t) where l is a po_s edge of the aeg and t is a type of fence. The first set potential-places is the set of such pairs that can be inserted into the program to forbid the cycles. The second set actual-places is the set of such pairs that have been set to 1 by our ILP. For each pair (l, t) we have an ILP variable t_l (in $\{0, 1\}$). We output the set actual-places, as it represents the locations in the code in need of a fence and the type of fence to insert for each of them. We also output the total cost of all these insertions, i.e.,

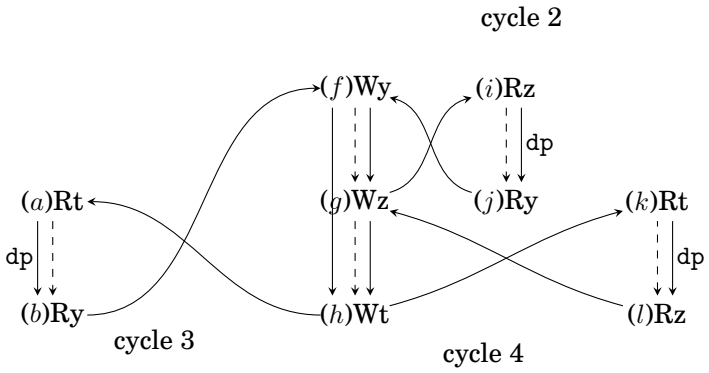
$$\sum_{(l,\mathsf{t})\in\text{potential-places}(C)} \mathsf{t}_l \times \mathsf{cost}(\mathsf{t}) \ .$$

The solver minimises this sum while satisfying the constraints.

6.3. Constraints in the ILP

We want to forbid all the cycles in the set that we are given after filtering, as explained in the preamble of this section. This requires placing an appropriate fence on each delay for each cycle in this set. Different delay pairs might need different fences, depending, e.g., on the directions (write or read) of their extremities. Essentially, we follow the table in Figure 7. For example, a write-read pair needs a full fence (e.g., mfence on x86, or sync on Power). A read-read pair can use anything amongst dependencies and fences. Our constraints ensure that we use the right type of fence for each delay pair.

[8] It is straight-forward to make the cost dependent on the location, and to obtain the information on how often a location is executed using standard profiling techniques.



$$
\begin{array}{ll}
\textbf{min} & 3 \cdot (\mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)}) + 2 \cdot (\mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)}) \\
\textbf{s.t.} & \text{cycle 2, delay } (f,g)\text{: } \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(f,g)} \geq 1 \\
& \text{cycle 3, delay } (f,h)\text{: } \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(g,h)} + \mathtt{lwf}_{(g,h)} \geq 1 \\
& \text{cycle 4, delay } (g,h)\text{: } \mathtt{f}_{(g,h)} + \mathtt{lwf}_{(g,h)} \geq 1
\end{array}
$$

Fig. 18. Variation of the example in Figure 16. We restrict to 3 cycles, (f) and (h) become writes, (a), (b), (j) and (k) become reads.

Inequalities as constraints. We first assume that all the program order delays are in po_s and we ignore Power and ARM special features (dependencies, control fences and communication delays). This case deals with relatively strong models, ranging from TSO to RMO. We relax these assumptions below.

In this setting, potential-places(C) is the set of all the po_s delays of the cycles in C. We ensure that every delay pair for every execution is fenced, by placing a fence on the static po_s edge for this pair, and this for each cycle given as input. Thus, we need at least one constraint per static delay pair d in each cycle.

If d is of the form poWR, as (g,h) in Figure 16 (cycle 4), only a full fence can fix it (cf. Figure 7), thus we impose $\mathsf{f}_d \geq 1$. If d is of the form poRR, as (f,h) in Figure 16 (cycle 3), we can choose any type of fence, i.e., $\mathsf{dp}_d + \mathsf{cf}_d + \mathsf{lwf}_d + \mathsf{f}_d \geq 1$.

Our constraints cannot be equalities because it is not certain that the resulting system would be satisfiable. To see this, suppose our constraints were equalities, and consider Figure 18, which is the same aeg as in Figure 16 but we limit ourselves to cycles 2, 3 and 4; (f) and (h) become writes while (a), (b), (j) and (k) become reads. For these cycles, we generate the following constraints:

(i) $\mathsf{lwf}_{(f,g)} + \mathsf{f}_{(f,g)} = 1$ for the delay (f,g) in cycle 2,
(ii) $\mathsf{lwf}_{(f,g)} + \mathsf{f}_{(f,g)} + \mathsf{lwf}_{(g,h)} + \mathsf{f}_{(g,h)} = 1$ for the delay (f,h) in cycle 3, and
(iii) $\mathsf{lwf}_{(g,h)} + \mathsf{f}_{(g,h)} = 1$ for the delay (g,h) in cycle 4.

By adding up **(i)** and **(iii)**, we obtain $\mathsf{lwf}_{(f,g)} + \mathsf{f}_{(f,g)} + \mathsf{lwf}_{(g,h)} + \mathsf{f}_{(g,h)} = 2$, which contradicts **(ii)**. By using inequalities, we allow several fences to on the same edge. In fact, the constraints only ensure the soundness; the optimality is achieved using the cost function.

Delays. The delays are in po_s^+ (and not always in po_s): in Figure 16, the delay (e,g) in cycle 1 does not belong to po_s but to po_s^+. Thus given a po_s^+ delay (x,y), we consider all the po_s pairs which appear between x and y, i.e.: $\mathsf{between}(x,y) \triangleq \{(e_1,e_2) \in \mathsf{po}_s \mid (x,e_1) \in \mathsf{po}_s^* \wedge (e_2,y) \in \mathsf{po}_s^*\}$. For example in Figure 16, we have $\mathsf{between}(e,g) = \{(e,f),(f,g)\}$. Thus, ignoring the use of dependencies and control fences for now, for the delay (e,g) in Figure 16, we will not impose $\mathsf{f}_{(e,g)} + \mathsf{lwf}_{(e,g)} \geq 1$ but

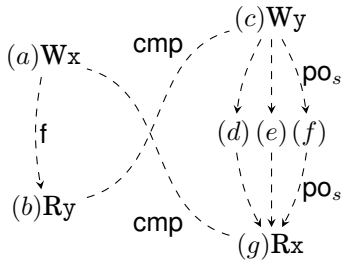


Fig. 19. Cycles sharing the edge (a, b).

rather $\mathsf{f}_{(e,f)} + \mathsf{lwf}_{(e,f)} + \mathsf{f}_{(f,g)} + \mathsf{lwf}_{(f,g)} \geq 1$. Indeed, a full fence or a lightweight fence in (e, f) or (f, g) will prevent the delay in (e, g).

Dependencies. Unlike for fences, it is not sufficient to place a dependency anywhere between two abstract events. Consider e and g in Figure 16: $\mathsf{dp}_{(e,f)}$ or $\mathsf{dp}_{(f,g)}$ would not fix the delay (e, g), but simply maintain the pairs (e, f) or (f, g), leaving the pair (e, g) free to be reordered. Thus if we choose to synchronise (e, g) using dependencies, we actually need a dependency from e to g: $\mathsf{dp}_{(e,g)}$. Dependencies only apply to pairs that start with a read; thus for each such pair (see the poRW and poRR cases in Figure 17), we add a variable for the dependency: (e, g) will be fixed with the constraint $\mathsf{dp}_{(e,g)} + \mathsf{f}_{(e,f)} + \mathsf{lwf}_{(e,f)} + \mathsf{f}_{(f,g)} + \mathsf{lwf}_{(f,g)} \geq 1$.

Control fences. Placing control fences after a conditional branch (e.g., bne on Power) prevents reads after this branch (see Figure 7) to be speculated. Thus, when building the aeg, we built a set poC for each branch, which gathers all the pairs of abstract events such that the first one is the last event before a branch, and the second is the first event after that branch. We can place a control fence before the second component of each such pair, if the second component is a read. Thus, we add cf_e as a possible variable to the constraint for read-read pairs (see the poRR case in Figure 17, where $\mathsf{ctrl}(d) = \mathsf{between}(d) \cap \mathsf{poC}$).

Cumulativity. For architectures like Power, where stores are non-atomic, we need to look for program order pairs that are connected to an external read-from (e.g., (c, d) in Figure 4 has an rf connected to it via event c). In such cases, we need to use a *cumulative fence* (e.g., lwsync or sync).

The locations to consider in such cases are: before (in po_s) the write w of the rfe, or after (in po_s) the read r of the rfe, i.e., $\mathsf{cumul}(w, r) = \{(e_1, e_2) \mid (e_1, e_2) \in \mathsf{po}_s \wedge ((e_2, w) \in \mathsf{po}_s^* \vee (r, e_1) \in \mathsf{po}_s^*)\}$. In Figure 16 (cycle 2), (g, i) over-approximates an rfe edge, and the edges where we can insert fences are in $\mathsf{cumul}(g, i) = \{(f, g), (i, j)\}$.

We need a cumulative fence as soon as there is a potential rfe, even if the adjacent po_s pairs do not form a delay. For example in Figure 4, suppose there is a dependency between the reads on T_1, and a fence maintaining write-write pairs on T_0. In that case we need to place a cumulative fence to fix the rfe, even if the two po_s pairs are themselves fixed. Thus, we quantify over all po_s pairs when we need to place cumulative fences. As only f and lwf are cumulative, we have

$$\begin{aligned}\mathsf{potential\text{-}places}(C) \triangleq \{(l,t) \mid\ & (t \in \{\mathsf{dp}\} \wedge\ l \in \mathsf{delays}(C)) \\ \vee\ & (t \in \mathbb{T} \backslash \{\mathsf{dp}\} \wedge\ l \in \textstyle\bigcup_{d \in \mathsf{delays}(C)} \mathsf{between}(d)) \\ \vee\ & (t \in \{\mathsf{f}, \mathsf{lwf}\} \wedge\ l \in \mathsf{po}_s(C))\}\ .\end{aligned}$$

6.4. Comparison with trencher

We illustrate the difference between trencher [Bouajjani et al. 2013] and our approach using Figure 19. There are three cycles that share the edge (a, b). They differ in the path taken between nodes c and g. Suppose that the user has inserted a full fence between a and b. To forbid the three cycles, we need to fence the thread on the right.

The trencher algorithm first calculates which pairs can be reordered: in our example, these are (c, g) via d, (c, g) via e and (c, g) via f. It then determines at which locations a fence could be placed. In our example, there are 6 options: (c, d), (d, g), (c, e), (e, g), (c, f), and (f, g). The encoding thus uses 6 variables for the fence locations. The algorithm then gathers all the *irreducible* sets of locations to be fenced to forbid the delay between c and g, where "irreducible" means that removing any of the fences would prevent this set from fully fixing the delay. As all the paths that connect c and g have to be covered, trencher needs to collect all the combinations of one fence per path. There are two locations per path, leading to 2^3 sets. Consequently, as stated in [Bouajjani et al. 2013], trencher needs to construct an exponential number of sets.

Each set is encoded in the ILP with one variable. For this example, trencher thus uses $6 + 8$ variables. It also generates one constraint per delay (here, 1) to force the solver to pick a set, and eight constraints to enforce that all the location variables are set to 1 if the set containing these locations is picked.

By contrast, musketeer only needs six variables: the possible locations for fences. We detect three cycles, and generate only three constraints to fix the delay. Thus, on a parametric version of the example, trencher's ILP grows exponentially whereas musketeer's is linear-sized.

7. IMPLEMENTATION, EXPERIMENTS, AND RUNTIME IMPACT

We implemented our new method, in addition to all the methods described in Section 2, in our tool musketeer. We use glpk (http://www.gnu.org/software/glpk) as the ILP solver. Once the locations and types of fences have been inferred by musketeer, the insertion of the fence instructions into the program source code is performed by a script. We explain how this is done in Section 7.1.

We compare our method and the pensieve method that we reimplemented to the existing tools listed in Section 3 on classic examples from the literature in Section 7.2. We also ran musketeer on executables from the Debian GNU/Linux distribution. We finally check the impact on runtime of the fences inferred and inserted in memcached, pfscan, dnshistory, and weborf. The results are reported in Section 7.3.

7.1. Inserting synchronisation into C code

Given an aeg, we output the static program order edges and the types of fences that should be placed there to forbid the critical cycles. This information is used by the fence insertion component to insert the fences into the C code. A static program order edge corresponds to two code locations (the source and the target of the edge) that access memory. We identify such code locations by a line number and the C expression on that line that corresponds to the access.

Our tool musketeer outputs a list of five-tuples with each tuple indicating a type of fence and where to place it. For example, for the code portion on the left of Figure 20, to indicate that an `mfence` should be placed between the read of `x` on line 1 and the read of `y` on line 3 (corresponding to an aeg edge $(Rx, Ry) \in \mathsf{po}_s$), musketeer would output the tuple (`mfence`, 1, `x`, 3, `y`). We then have some freedom of choice where to place the fence, as shown in the middle and on the right of Figure 20. We insert fences via inline

```
1 r1 = x;
2 r2 = r1+2;
3 r3 = y;
4
```

```
1 r1 = x;
2 asm (”mfence”);
3 r2 = r1+2;
4 r3 = y;
```

```
1 r1 = x;
2 r2 = r1+2;
3 asm (”mfence”);
4 r3 = y;
```

Fig. 20. Choices for placing a fence.

```
1  void* t0(void* arg) {
2      int r;
3      if (x > 0)
4        r = y;
5  }
6
7
8
9
10
```

```
1  void* t0(void* arg) {
2      int r;
3      if (x > 0) {
4        asm (”dmb”);
5        r = y;
6      }
7  }
8
9
10
```

```
1    ldr    r3, [r3]
2    cmp    r3, #0
3    ble    .L2
4    dmb
5    movw r3, #:lower16:y
6    movt r3, #:upper16:y
7    ldr    r3, [r3]
8    str    r3, [r7, #4]
9  .L2:
```

Fig. 21. Original program (left), program with fence inserted (middle), program compiled to ARM assembly (right).

assembly statements, which we give in abbreviated form as `asm ("code")`.[9] We next illustrate how we insert fences and dependencies in a piece of C code.

Fences. We insert fences into a program via inline assembly. In Figure 21, we insert a fence between the read of `x` in line 3 and the read of `y` in line 4. Adding the inline assembly statement also requires us to add curly braces to the if statement in order to ensure that both the fence and the read of `y` are performed conditionally. The program compiled to ARM assembly (with `gcc -O0`), and the result is given on the right.

Dependencies. To insert dependencies, we further need to rewrite the code. Consider a read-read pair, corresponding to lines 3 and 4 on the left of Figure 22. We enforce an *address dependency* from the read of `x` to the read of `y`, by using a register (r3) to perform some computation which always returns `0` (in this case XOR-ing a register with itself), then add this result to the address of y. This transformation ensures that the compiled ARM assembly (`gcc -O0`; given on the right of Figure 22) has an address dependency between the read of `x` and the read of `y`.

7.2. Experiments and benchmarks

Our tool analyses C programs. The dfence tool also handles C code, but requires some high-level specification for each program, which was not available to us. The memorax tool works on a process-based language that is specific to the tool. The offence tool works on a subset of assembler for x86, ARM and Power. The pensieve tool originally handled Java, but we did not have access to it and have therefore re-implemented the method. Similarly, remmex handles Promela-like programs and trencher analyses transition systems. Most of the tools come with some of the benchmarks in their own languages; not all benchmarks were however available for each tool. We have re-implemented some of the benchmarks for offence.

[9]We also give the `volatile` keyword and the `"memory"` clobber to ensure that the inline assembly statements are not optimised: `asm volatile("code":::"memory")`. See https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html.

```
1  void* t0(void* arg) {
2    int r1, r2;
3    r1 = x;
4    r2 = y;
5  }
6
7
8
9
10
```

```
1  void* t0(void* arg) {
2    int r1, r2;
3    r1 = x;
4    int r3;
5    asm (
6      "eors %0,%1,%1"
7      :"=r"(r3)
8      :"r"(r1));
9    r2 = *(&y + r3);
10 }
```

```
1  t0:
2    movw r3,#:lower16:x
3    movt r3,#:upper16:x
4    ldr  r2,[r3, #0]
5    eors r2, r2,r2
6    movw r3,#:lower16:y
7    movt r3,#:upper16:y
8    ldr  r3,[r3,r2,lsl #2]
9    bx   lr
10
```

Fig. 22. Original program (left), program with dependency inserted (middle), program compiled to ARM assembly (right).

	Dek		Pet		Lam		Szy		Par	
LoC	50		37		72		54		96	
dfence	–	–	–	–	–	–	–	–	–	–
memorax	0.4	2	1.4	2	79.1	4	1.3	3	1.1	0
musketeer	**0.0**	**5**	**0.0**	**3**	**0.0**	**8**	**0.0**	**8**	**0.0**	**3**
offence	0.0	2	0.0	2	0.0	8	0.0	8	–	–
pensieve	0.0	16	0.0	6	0.0	24	0.0	22	0.0	7
remmex	0.5	2	0.5	2	2.0	4	1.8	5	–	–
trencher	1.6	2	1.3	2	1.7	4	1.1	8	0.5	1
persist	0.8	2	0.7	2	2.5	8	2.1	8	0.9	1

Fig. 23. All tools on the CLASSIC series for TSO.

We now report our experimental data. CLASSIC and FAST gather examples from the literature and related work. The DEBIAN benchmarks are packages of the Debian GNU/Linux distribution version 7.1. CLASSIC and FAST were run on a x86-64 Intel Core2 Quad Q9550 machine with 4 cores (2.83 GHz) and 4 GB of RAM. DEBIAN was run on a x86-64 Intel Core i5-3570 machine with 4 cores (3.40 GHz) and 4 GB of RAM.

CLASSIC. This set consists of Dekker's mutex (Dek) [Dijkstra 1965]; Peterson's mutex (Pet) [Peterson 1981]; Lamport's fast mutex (Lam) [Lamport 1987]; Szymanski's mutex (Szy) [Szymanski 1988]; and Parker's bug (Par) [Dice 2009]. We ran all tools in this series for TSO (the model common to all). For each example, Figure 23 gives the number of fences inserted, and the time (in seconds) needed. When an example is not available in the input language of a tool, we write "–". We used memorax with the option `-o1`, to compute one *maximal permissive* set and not all. For remmex on Szymanski, we give the number of fences found by default (which may be non-optimal). Its "maximal permissive" option lowers the number to 2, at the cost of a slow enumeration. We observe that our tool musketeer is less precise than most tools, but outperforms them in terms of performance.

FAST. This set consists of Cil, Cilk 5 Work Stealing Queue (WSQ) [Frigo et al. 1998]; CL, Chase-Lev WSQ [Chase and Lev 2005]; Fif, Michael et al.'s FIFO WSQ [Michael et al. 2009]; Lif, Michael et al.'s LIFO WSQ [Michael et al. 2009]; Anc, Michael et al.'s Anchor WSQ [Michael et al. 2009]; Har, Harris' set [Detlefs et al. 2000]. For each example and tool, Figure 24 gives the number of fences inserted (under TSO) and the time needed to do so. For dfence, we used the setting of Liu et al. [2012]: the tool has up to 20 attempts to find fences. We were unable to apply dfence on some of the FAST examples: we thus reproduce the number of fences given in [Liu et al. 2012], and write

	Cil		CL		Fif		Lif		Anc		Har	
LoC	97		111		150		152		188		179	
dfence	7.8	3	6.2	3	$\sim$	0	$\sim$	0	$\sim$	0	$\sim$	0
musketeer	**0.0**	**3**	**0.0**	**1**	**0.1**	**1**	**0.0**	**1**	**0.1**	**1**	**0.6**	**4**
pensieve	0.0	14	0.0	8	0.1	33	0.0	29	0.0	44	0.1	72
trencher	0.5	1	8.6	3	–	–	–	–	–	–	–	–
persist	1.3	2	2.3	1	1.0	1	14.6	1	0.9	1	–	–

Fig. 24. Tools on the FAST series for TSO.

			TSO		Power	
	LoC	nodes	fences	time (s)	fences	time (s)
memcached	9944	571	4	12.4	61	83.9
lingot	2504	110	0	0.3	0	0.3
weborf	1915	432	1	94.3	1	95.7
timemachine	977	78	0	0.1	0	0.1
see	2227	302	0	2.4	0	2.3
blktrace	1438	416	0	1.9	5	16.5
ptunnel	1119	1805	3	19.6	32	43.3
proxsmtpd	2023	1934	0	167.8	–	timeout
ghostess	2275	1008	0	1.9	0	1.8
dnshistory	1358	888	1	5.9	7	25.7

Fig. 25. musketeer on selected benchmarks of the DEBIAN series for TSO and Power.

$\sim$ for the time. We applied musketeer to this series, for all architectures. The fencing times for TSO and Power are almost identical, except for the largest example, namely Har ($0.1\,\mathrm{s}$ vs. $0.6\,\mathrm{s}$).

Similar experiments to our CLASSIC and FAST series have been performed by Abdulla et al. [2015]. On some examples their tool persist could infer fences while musketeer timed out. These were cases where a higher number of threads was used (e.g., Peterson's algorithm with 4 threads as opposed to 2 threads as in our experiments). They also evaluated the scalability of persist, trencher, and musketeer on 8 examples when the number of threads was increased. They found that persist and musketeer scaled better than the other on 4 examples each.

DEBIAN. This set consists of in total 715 goto-programs that have been built from Debian GNU/Linux 7.1 (details of this process are described in [Kroening and Tautschnig 2014]). The compiled goto-programs can be found at http://theory.eecs.qmul.ac.uk/debian+mole/pkgs/. A small excerpt of our results is given in Figure 25. The full data set is provided at http://www.cprover.org/wmm/musketeer. For each program, we give the lines of code and number of nodes in the aeg.

We used musketeer on these programs to demonstrate its scalability and its ability to handle deployed code. Most programs already contain fences or operations that imply them, such as compare-and-swaps or locks. Our tool musketeer takes these fences into account and infers a set of additional fences sufficient to guarantee SC. The largest program we handle is memcached ($\sim$10,000 LoC). Our tool needs $12.4\,\mathrm{s}$ to place fences for TSO, and $83.9\,\mathrm{s}$ for Power.

A more meaningful measure for the difficulty of an instance is the number of nodes in the aeg. For example, proxsmtpd has 1934 nodes and 2023 LoC. The fencing takes $167.8\,\mathrm{s}$ for TSO, but times out for Power due to the number of cycles.

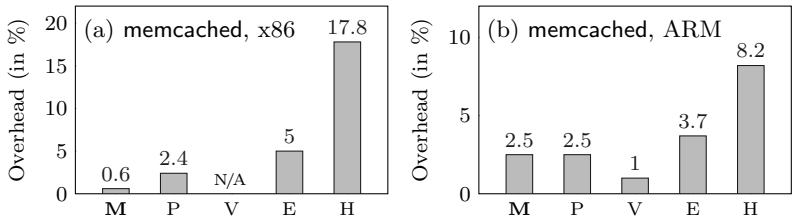


Fig. 26. Runtime overheads due to fences inserted in memcached for each strategy.

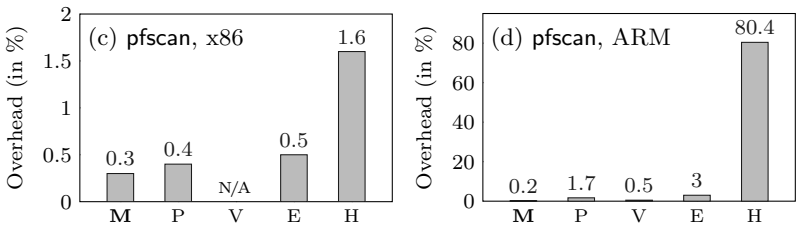


Fig. 27. Runtime overheads due to fences inserted in pfscan for each strategy.

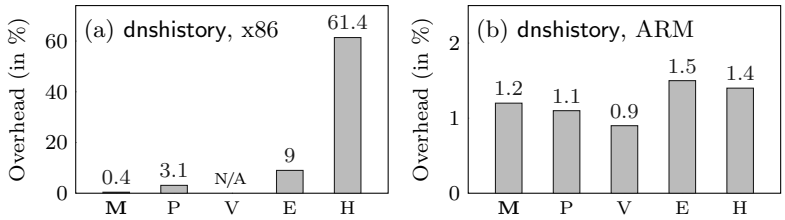


Fig. 28. Runtime overheads due to fences in inserted dnshistory and for each strategy.

Of the 715 binaries we ran musketeer on, for 598 binaries it could successfully infer fences for TSO, and for 579 binaries it could successfully infer fences for Power. Whenever musketeer could infer fences for Power (within the available time limit of 10m), it could also infer fences for TSO. On 117 binaries our tool reached a timeout for both TSO and Power (timeout 10m). On 57 of those binaries it timed out during the pointer analysis phase, while on the remaining 60 binaries it timed out during the cycle search in the aeg. Our tool never timed out during the ILP solving phase on the Debian benchmarks.

Not all fences inferred by musketeer are necessary to enforce SC, due to the imprecision introduced by the aeg abstraction. We discuss the execution time overhead of the program versions with fences inserted by musketeer next.

7.3. Impact on runtime

We finally measured the impact of fences for the programs memcached (an in-memory caching system), pfscan (a file scanning tool), dnshistory (a DNS lookup tool), and weborf (a lightweight web server), with experiments similar to those in Section 2. We built new versions of the programs according to the fencing strategies described in Section 2.

The workload for the memcached daemon was generated using the memtier benchmarking tool. We killed the memcached daemon after $60\,\mathrm{s}$. The pfscan tool was used to concurrently search for a short string (5 characters) in several large text files. The dnshistory tool was invoked to perform a series of lookups concurrently from several

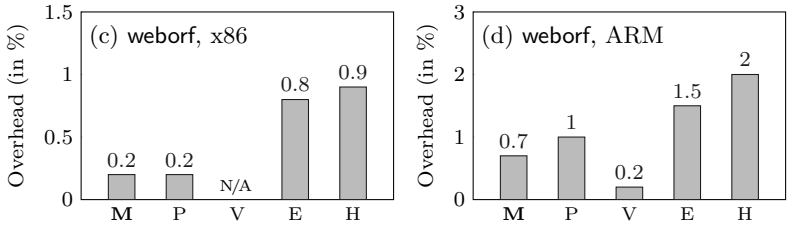


Fig. 29. Runtime overheads due to inserted fences in weborf for each strategy.

	memcached on x86	memcached on ARM	pfscan on x86	pfscan on ARM
(M)	[29.781; 29.893]	[11.513; 11.615]	[15.074; 15.097]	[19.112; 19.139]
(P)	[30.317; 30.423]	[11.497; 11.628]	[15.083; 15.107]	[19.401; 19.427]
(V)	N/A	[11.324; 11.462]	N/A	[19.176; 19.204]
(E)	[31.093; 31.169]	[11.633; 11.762]	[15.086; 15.118]	[19.659; 19.684]
(H)	[34.904; 34.947]	[12.059; 12.365]	[15.270; 15.293]	[34.422; 34.457]

Fig. 30. Confidence intervals for the mean execution times (in sec) for memcached and pfscan.

	dnshistory on x86	dnshistory on ARM	weborf on x86	weborf on ARM
(M)	[1.631; 1.649]	[1.193; 1.209]	[1.102; 1.113]	[2.022; 2.060]
(P)	[1.676; 1.704]	[1.193; 1.209]	[1.099; 1.116]	[2.030, 2.068]
(V)	N/A	[1.192; 1.205]	N/A	[2.012; 2.052]
(E)	[1.779; 1.794]	[1.196; 1.214]	[1.110; 1.119]	[2.042; 2.074]
(H)	[2.633; 2.661]	[1.197; 1.211]	[1.110; 1.122]	[2.050; 2.089]

Fig. 31. Confidence intervals for the mean execution times (in sec) for dnshistory and weborf.

threads. The workload for the weborf server was generated by a script that performs a series of http requests to the server.

The results are shown in Figures 26–29. The bars indicate the mean overhead w.r.t. the original, unmodified programs. The corresponding confidence intervals for the mean runtimes are shown in Figures 30–31.

Overall, the overhead of fences was most noticeable for memcached on both x86 and ARM. Adding a fence after every access to static or heap data yielded overheads of up to 17.8%. Adding fences via an escape analysis yielded overheads to up to 5%. Amongst the approaches that guarantee SC (i.e., all but V), the best results were achieved with our tool musketeer.

8. CONCLUSIONS

We introduced a novel method for deriving a set of fences that restore sequential consistency on architectures that implement TSO and the IBM Power consistency models. We have implemented the method in a new tool called musketeer. We compared it to existing tools and observed that it outperforms them. We demonstrated on our DEBIAN series that musketeer can handle deployed code, with a large potential for scalability.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. 2013. Memorax, a Precise and Sound Tool for Automatic Fence Insertion under TSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*. Springer, 530–536.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lang, and Tuan Phong Ngo. 2015. Precise and Sound Automatic Fence Insertion Procedure under PSO. In *NETYS*.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015. The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO. In *European Symposium on Programming on Programming (ESOP)*. Springer, 308–332.

Sarita V. Adve and Kourosh Gharachorloo. 1995. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1995), 66–76.

Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. 2011. Soundness of Data Flow Analyses for Weak Memory Models. In *Programming Languages and Systems (APLAS) (LNCS)*, Vol. 7078. Springer, 272–288.

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software Verification for Weak Memory via Program Transformation. In *European Symposium on Programming (ESOP) (LNCS)*. Springer, 512–532.

Jade Alglave and Luc Maranget. 2011. Stability in Weak Memory Models. In *Computer Aided Verification (CAV) (LNCS)*, Vol. 6806. Springer, 50–66.

Jade Alglave, Luc Maranget, S. Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification (CAV) (LNCS)*, Vol. 6174. Springer, 258–272.

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. DOI:http://dx.doi.org/10.1145/2627752

John Bender, Mohsen Lesani, and Jens Palsberg. 2015. Declarative Fence Insertion. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 367–385.

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *European Symposium on Programming (ESOP) (LNCS)*, Vol. 7792. Springer, 533–553.

Ahmed Bouajjani, R. Meyer, and E. Moehlmann. 2011. Deciding Robustness Against Total Store Ordering. In *Automata, Languages and Programming (ICALP) (LNCS)*, Vol. 6756. Springer, 428–440.

C11 2011. Information technology – Programming languages – C. In *BS ISO/IEC 9899:2011*.

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 21–28.

David Detlefs, Christine H. Flood, Alex Garthwaite, Paul A. Martin, Nir Shavit, and Guy L. Steele Jr. 2000. Even Better DCAS-Based Concurrent Deques. In *Distributed Computing (DISC) (LNCS)*, Vol. 1914. Springer, 59–73.

David Dice. 2009. (November 2009). https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park

Edsger W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (1965), 569.

Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing (ICS)*. ACM, 285–294.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. ACM, 212–223.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan and Kaufmann, Burlington.

Saurabh Joshi and Daniel Kroening. 2015. Property-Driven Fence Insertion Using Reorder Bounded Model Checking. In *FM*. Springer, 291–307.

Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static data race detection for concurrent programs with asynchronous calls. In *FSE*. ACM, 13–22.

Arvind Krishnamurthy and Katherine A. Yelick. 1996. Analyses and Optimizations for Shared Address Space Programs. *J. Par. Dist. Comp.* 38, 2 (1996).

Daniel Kroening and Michael Tautschnig. 2014. Automating Software Analysis at Large Scale. In *Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS)*. Springer, 30–39.

Michael Kuperstein, Martin T. Vechev, and Eran Yahav. 2010. Automatic inference of memory fences. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 111–119.

Michael Kuperstein, Martin T. Vechev, and Eran Yahav. 2011. Partial-coherence abstractions for relaxed memory models. In *PLDI*. 187–198.

Leslie Lamport. 1979. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.* 46, 7 (1979).

Leslie Lamport. 1987. A Fast Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.* 5, 1 (1987).

Jaejin Lee and David A. Padua. 2001. Hiding Relaxed Memory Consistency with a Compiler. *IEEE Trans. Comput.* 50 (2001), 824–833.

Alexander Linden and Pierre Wolper. 2013. A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Vol. 7795. Springer, 339–353.

Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. 2012. Dynamic synthesis for relaxed memory models. In *PLDI*. ACM, 429–440.

Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *ISCA*. ACM, 388–400.

Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *PLDI*. ACM, 199–210.

Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. 2014. Synthesis of Memory Fences via Refinement Propagation. In *SAS*. Springer, 237–252.

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. ACM, 267–275.

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent work stealing. In *Principles and Practice of Parallel Programming (PPOPP)*. ACM, 45–54.

Vincent Nimal. 2015. *Static Analyses over Weak Memory*. Ph.D. Dissertation. University of Oxford.

Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 131–150.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics (TPHOLs) (LNCS)*, Vol. 5674. Springer, 391–407.

Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12, 3 (1981), 115–116.

2009. *Power ISA Version 2.06*.

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS* 10, 2 (1988), 282–312.

SPARC 1994. SPARC Architecture Manual Version 9. (1994).

Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. 2009. Reducing Memory Ordering Overheads in Software Transactional Memory. In *CGO*. IEEE, 13–24.

Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David A. Padua. 2005. Compiler techniques for high performance sequentially consistent Java programs. In *PPOPP*. ACM, 2–13.

Boleslaw K. Szymanski. 1988. A simple solution to Lamport's concurrent programming problem with linear wait. In *ICS*. 621–626.

Robert Tarjan. 1973. Enumeration of the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 2, 3 (1973), 211–216.

Viktor Vafeiadis and Francesco Zappa Nardelli. 2011. Verifying Fence Elimination Optimisations. In *Static Analysis (SAS) (LNCS)*, Vol. 6887. Springer, 146–162.

Glynn Winskel. 1986. Event Structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*. 325–392. DOI:http://dx.doi.org/10.1007/3-540-17906-2_31