Don't sit on the fence*

A static analysis approach to automatic fence insertion

Jade Alglave[1], Daniel Kroening[2], Vincent Nimal[2], and Daniel Poetzl[2]

[1]University College London [2]University of Oxford

Abstract 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 detail experiments on more than 350 executables of packages found in Debian Linux 7.1, e.g. memcached (about 10000 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*, *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) [20], a theoretical model where the execution of a program corresponds to an interleaving of 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.

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 Sec. 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 reported a bug found in a published C11 implementation of a work-stealing queue [27].

We address here the question of how to *synthesise* fences, i.e. automatically place them in a program to enforce robustness/stability [9,5] (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

* Supported by SRC/2269.002, EPSRC/H017585/1 and ERC/280053.

queues that implement the features of the hardware. Similarly, a good fraction of the fence synthesis methods, e.g. [23,18,19,24,3,10] (see also Fig. 2), rely on operational models to describe executions of programs.

Challenges Thus, methods using operational models inherit the limitations of methods based on interleavings, *e.g.* the *"severely limited scalability"*, as [24] puts 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 à la Power makes the problem significantly more difficult. Intel x86 offers only one fence (mfence), but Power offers a variety of synchronisation: fences (e.g. sync and lwsync), or 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; 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 [6]. 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. [30,5,9,10], 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 Our method is sound for a wide range of architectures, including x86-TSO, Power and ARM; and scales for large code bases, such as memcached (about 10000 LoC). We implemented it in our new musketeer tool. Our method is the most precise of the static analysis methods (see Sec. 2). To do this comparison, we implemented all these methods in our tool; for example, the pensieve policy [32] was designed for Java only, and we now provide it for x86-TSO, Power and ARM. Thus, our tool musketeer gives a comparison point for the field.

Outline We discuss the performance impact of fences in Sec. 2, and survey related work in Sec. 3. We recall our weak memory semantics in Sec. 4. We detail how we detect critical cycles in Sec. 5, and how we place fences in Sec. 6. In Sec. 7, we compare existing tools and our new tool musketeer. We provide the sources, benchmarks and experimental reports online at http://www.cprover.org/wmm/musketeer.

2 Motivation

Before optimising the placement of fences, we investigated whether naive approaches to fence insertion indeed have a negative performance impact. To that end, we measured

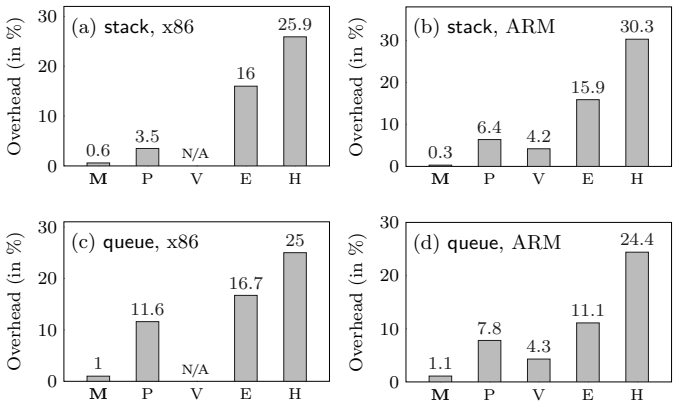


Fig. 1. Overheads for the different fencing strategies

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 built 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 [32], i.e. a static over-approximation of Shasha and Snir's eponymous (dynamic) analysis [30] (see also the discussion of Lee and Padua's work [22] in Sec. 3);
- (V) with fences following the *Visual Studio* policy, i.e. guaranteeing acquire/release semantics (in the C11 sense [2]), but not SC, for reads and writes of `volatile` variables (see http://msdn.microsoft.com/en-us/library/vstudio/jj635841.aspx, accessed 04-11-2013). On x86, no fences are necessary 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) *static global* or *heap data*.

We emphasise that 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 ran all versions 100 times, on an x86-64 Intel Core i5-3570 with 4 cores (3.40 GHz) and 4 GB of RAM, and on an ARMv7 (32-bit) Samsung Exynos 4412 with 4 cores (1.6 GHz) and 2 GB of RAM.

For each program version, Fig. 1 shows the mean overhead w.r.t. the unfenced program. We give the overhead in *user time* (as given by Linux `time`), i.e. the time spent by the program in user mode on the CPU. We refer the reader to our study of the statistical significance of these experiments (using confidence intervals) in the full version of this paper [8]. Amongst the approaches that guarantee SC (i.e. all but V), the best results were achieved with our tool musketeer.

3 Related work

authors	tool	model style	objective
Abdulla et al. [3]	memorax	operational	reachability
Alglave et al. [6]	offence	axiomatic	SC
Bouajjani et al. [10]	trencher	operational	SC
Fang et al. [15]	pensieve	axiomatic	SC
Kuperstein et al. [18]	fender	operational	reachability
Kuperstein et al. [19]	blender	operational	reachability
Linden et al. [23]	remmex	operational	reachability
Liu et al. [24]	dfence	operational	specification
Sura et al. [32]	pensieve	axiomatic	SC

Fig. 2. Fence synthesis tools

The work of Shasha and Snir [30] is a foundation for 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 SC. Fig. 2 classifies the methods mentioned in this section w.r.t. their style of model (operational or axiomatic). We report our experimental comparison of these tools in Sec. 7. Below, we detail fence synthesis methods per style. We write TSO for Total Store Order, implemented in Sparc TSO [31] and Intel x86 [28]. We write PSO for Partial Store Order and RMO for Relaxed Memory Order, two other Sparc architectures. We write Power for IBM Power [1].

Operational models Linden and Wolper [23] explore all executions (using what they call *automata acceleration*) to simulate the reorderings occuring under TSO and PSO. Abdulla et al. [3] 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.

Kuperstein et al. [18] 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 [19] 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. Liu et al. [24] 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. [10] build on an operational model of TSO. They look for *minimum violations* (viz. critical cycles) by enumerating *attackers* (viz. delays). Like us, they use linear programming. 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.

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

Axiomatic models Krishnamurthy et al. [17] 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.

Lee and Padua [22] 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. [15] in pensieve, a compiler for Java. Sura et al. later implemented a more precise approach in pensieve [32] (see (P) in Sec. 2). They pair the cycle detection with an analysis to detect synchronisation that could prevent cycles.

Alglave and Maranget [6] revisit Shasha and Snir for contemporary memory models and insert fences following a refinement of [22]. Their offence tool handles snippets of assembly code only, where the memory locations need to be explicitly given.

Others We cite the work of Vafeiadis and Zappa Nardelli [35], who present an optimisation of the certified CompCert-TSO compiler to remove redundant fences on TSO. Marino et al. [25] experiment with an SC-preserving compiler, showing overheads of no more than $34\,\%$. Nevertheless, they emphasise that *"the overheads, however small, might be unacceptable for certain applications"*.

4 Axiomatic memory model

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

To describe such situations, we use the framework of [6], embracing in particular SC, Sun TSO (i.e. the x86 model [28]), and a fragment of Power. The core of this framework consists of *relations* over *memory events*.

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`

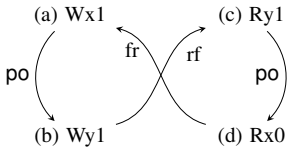


Fig. 3. Message Passing (**mp**)

We illustrate this framework using a *litmus test* (Fig. 3). The top shows a multi-threaded program. The shared variables `x` and `y` are assumed to be initialised to zero. A store instruction (e.g. `x` $\leftarrow$ `1` on T_0) gives rise to a write event ((a)Wx1), and a load instruction (e.g. `r1` $\leftarrow$ `y` on T_1) to a read event ((c)Ry1). The bottom of Fig. 3 shows one particular execution of the program (also called *event graph*), corresponding to the final state `r1=1` and `r2=0`.

In the framework of [6], an execution that is not possible on SC has a cyclic event graph (as the one shown in Fig. 3). A weaker architecture may *relax* some of the relations contributing to a cycle. If the removal of the relaxed edges from the event graph makes it acyclic, the architecture allows the execution. For example, Power relaxes the program order po (amongst other things), thereby making the graph in Fig. 3 acyclic. Hence, the given execution is allowed on Power.

Formalisation An *event* is a memory read or a write to memory, composed of a unique identifier, a direction (R for read or W for write), a memory address, and a value. We

represent each instruction by the events it issues. In Fig. 3, we associate the store instruction $x \leftarrow 1$ in thread T_0 with the event $(a)\mathrm{W}x1$.

A set of events $\mathbb{E}$ and their program order po form an *event structure* $E \triangleq (\mathbb{E}, \mathsf{po})$. The program order po is a per-thread total order over $\mathbb{E}$. 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 when the value written by the store was computed from the value obtained by the load.

We represent the *communication* between threads via an *execution witness* $X \triangleq (\mathsf{co}, \mathsf{rf})$, which consists of two relations over the events. First, the *coherence* co is a per-address total order on write events which models the *memory coherence* widely assumed by modern architectures. It links a write w to any write w' to the same address that hits the memory after w. Second, the *read-from* relation rf links a write w to a read r such that r reads the value written by w. Finally, we derive the *from-read* relation fr from co and rf. 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: $(r, w) \in \mathsf{fr} \triangleq \exists w'.(w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{co}$.

In Fig. 3, the specified outcome corresponds to the execution below if each location initially holds 0. If `r1=1` in the end, the read (c) on T_1 took its value from the write (b) on T_0, hence $(b, c) \in \mathsf{rf}$. If `r2=0` in the end, the read (d) took its value from the initial state, thus before the write (a) on T_0, hence $(d, a) \in \mathsf{fr}$. In the following, we write rfe (resp. $\mathsf{coe}, \mathsf{fre}$) for the *external read-from* (resp. coherence, from-read), i.e. when the source and target belong to different threads.

Relaxed or safe When a thread can read from its own store buffer [4] (the typical TSO/x86 scenario), we relax the internal read-from, 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 [4]), 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 on TSO/x86. Some program-order pairs may be relaxed (e.g. write-read pairs on x86, and all but dp ones on Power), i.e. only a subset of po is guaranteed to occur in order. This subset constitutes the *preserved program order*, ppo. When a relation must not be relaxed on a given architecture, we call it *safe*.

	SC	x86	Power
poWR	yes	mfence	sync
poWW	yes	yes	sync, lwsync
poRW	yes	yes	sync, lwsync, dp
poRR	yes	yes	sync, lwsync, dp, branch;isync

Fig. 4. ppo and fences per architecture

Fig. 4 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 "yes" 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 [6], the relation fence (with $\mathsf{fence} \subseteq \mathsf{po}$) 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, even though rfe alone would not be safe. In Fig. 3, 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.

Architectures An *architecture* A determines the set safe_A of relations safe on A. Following [6], we always consider the coherence co, the from-read relation fr and the fences to be safe. SC relaxes nothing, i.e. rf and po are safe. TSO authorises the reordering of write-read pairs and store buffering but nothing else.

Critical cycles Following [30,5], for an architecture A, a *delay* is a po or rf edge that is not safe (i.e. is relaxed) on A. An execution (E, X) is valid on A yet not on SC iff it contains critical cycles [5]. Formally, a *critical cycle* w.r.t. A is a cycle in $\mathsf{po} \cup \mathsf{com}$, where $\mathsf{com} \triangleq \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ is the *communication relation*, which has the following characteristics (the last two ensure the minimality of the critical cycles): (1) the cycle contains at least one delay for A; (2) per thread, (i) there are at most two accesses a and b, and (ii) they access distinct memory locations; and (3) for a memory location ℓ, there are at most three accesses to ℓ along the cycle, which belong to distinct threads.

Fig. 3 shows a critical cycle w.r.t. Power. The po edge on T_0, the po edge on T_1, and the rf edge between T_0 and T_1, are all unsafe on Power. On the other hand, the cycle in Fig. 3 does not contain a delay w.r.t. TSO, and is thus not a critical cycle on TSO.

To forbid executions containing critical cycles, one can insert fences into the program to prevent delays. To prevent a po delay, a fence can be inserted between the two accesses forming the delay, following Fig. 4. To prevent an rf delay, a cumulative fence must be used (see Sec. 6 for details). For the example in Fig. 3, 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. We explained in Sec. 4 that we should place fences along the critical cycles of the program executions. To find the critical cycles, we look for cycles in an over-approximation of all the executions of the program. We hence avoid enumeration of all traces, which would hinder scalability, and get all the critical cycles of all program executions at once. Thus we can find all fences preventing the critical cycles corresponding to two executions in one step, instead of examining the two executions separately.

To analyse a C program, e.g. on the left-hand side of Fig. 5, we convert it to a *goto-program* (right-hand side of Fig. 5), the internal representation of the CProver framework; we refer to http://www.cprover.org/goto-cc for details. 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 [7]. A full explanation of how we handle pointers is available in [8]. The C program in Fig. 5 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.

```
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
```

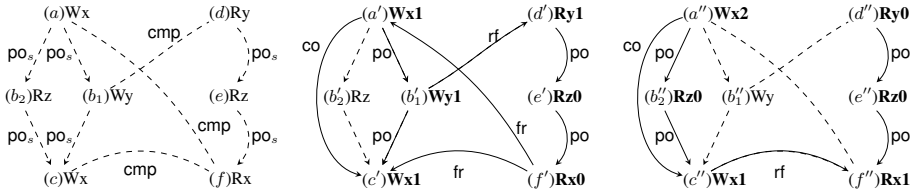
```
thread_2
    int r2, r3, r4;
    r2 = y;
    r3 = z;
    r4 = x;
    end_function
```

Fig. 5. A C program (left) and its goto-program (right)

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), shown in Fig. 6(a). The events a, b_1, b_2 and c (resp. d, e and f) correspond to thread$_1$ (resp. thread$_2$) in Fig. 5. 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 executions of a program (in the sense of Sec. 4). Fig. 6(b) and (c) give two executions associated with the aeg shown in Fig. 6(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 Fig. 6(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 Fig. 6(b) and (c)). The execution in Fig. 6(b) has a critical cycle (with respect to e.g. Power) between the events a', b'_2, d', and f'. The execution in Fig. 6(c) does not have a critical cycle.

Full details of the construction of the aegs from goto-programs, including a semantics of goto-programs in terms of abstract events, are available in the full version of this paper [8]. Function calls are inlined for better precision. Currently, the implementation does not handle recursion.



(a) aeg of Fig. 5 (b) ex. with critical cycle (c) ex. without critical cycle

Fig. 6. The aeg of Fig. 5 and two executions corresponding to it

Loops and arrays We explain how to deal with loops statically. If we build our aeg directly following the cfg, with a po_s back-edge connecting the end of the body to its entry, we already handle most of the cases. Recall from Sec. 4 that in a critical cycle (2.i) there are two events per thread, and (2.ii) two events on the same thread target two 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). The critical cycle will be trivially captured by its static counterpart that abstracts in particular these events with 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) of a later iteration j (i.e., $i \leq j$), then these events are abstracted respectively by (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 and the transitivity of our cycle search, 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.

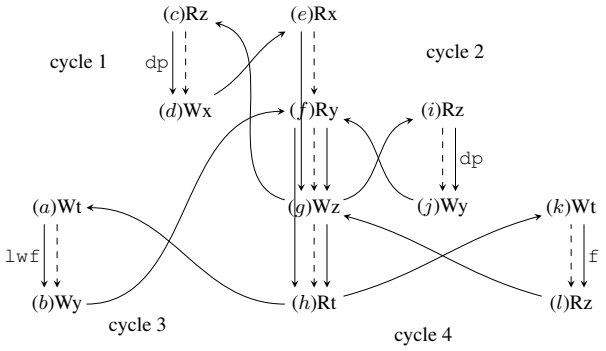
The only case that is not handled by this approach is when (a_i) and (b_j) are abstracted by the same abstract event, say (c). As the variables addressed by the events on the same thread of a cycle need to be different, this case can only occur when (a_i) and (b_j) are accessing an array or a pointer 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, we copy the body of the loop and do not add a po_s back-edge. 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). The back-edge is no longer necessary, as the abstract events reachable through this back-edge are replicated in the second body. Thus, all the previous cases are also covered.

We have implemented the duplication of the loop bodies only for loops that contain accesses to arrays. In case of nested loops, we ensure that we duplicate each of the sub-bodies only once in order to avoid an exponential explosion. This approach is again sufficient owing to the maximum of two events per thread in a critical cycle and the transitivity of po.

Pointers We explain how to deal with the varying imprecision of pointer analyses in a sound way. If we have a precise pointer analysis, we insert as many abstract events as required for the objects pointed to. Similarly to array accesses, a pointer might refer to two separate memory locations dynamically, e.g., if pointer arithmetic is used. If such an access is detected inside a loop, the body is replicated as described above. If the analysis cannot determine the location of an access, we insert an abstract event accessing any shared variable. This event can communicate with any variable accessed in other threads.

Cycle detection Once we have the aeg, we enumerate (using Tarjan's algorithm [34]) its potential critical cycles by searching for cycles that contain at least one edge that is a delay, as defined in Sec. 4.



$$\textbf{min}\ \ \mathtt{dp}_{(e,g)} + \mathtt{dp}_{(f,h)} + \mathtt{dp}_{(f,g)} + 3 \cdot (\mathtt{f}_{(e,f)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)}) + 2 \cdot (\mathtt{lwf}_{(e,f)} + \mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)})$$

s.t. cycle 1, delay (e, g): $\mathtt{dp}_{(e,g)} + \mathtt{f}_{(e,f)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(e,f)} + \mathtt{lwf}_{(f,g)} \geq 1$

cycle 2, delay (f, g): $\mathtt{dp}_{(f,g)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(f,g)} \geq 1$

cycle 3, delay (f, h): $\mathtt{dp}_{(f,h)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)} + \mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)} \geq 1$

cycle 4, delay (g, h): $\mathtt{f}_{(g,h)} \geq 1$

Fig. 7. Example of resolution with between

6 Synthesis

In Fig. 7, we have 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 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. Thus the aeg of Fig. 7 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 amount of fences to forbid as many cycles as possible. For example, placing a full fence sync between f and g in Fig. 7 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 Fig. 7, 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.

We model our problem as an *integer linear program* (ILP) (see Fig. 8), which we explain in this section. 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

Input: aeg ($\mathbb{E}_s$,po$_s$,cmp) and potential critical cycles $C = \{C_1, ..., C_n\}$
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 actual-places(C) of pairs (l, t) s.t. t_l is set to 1 in the ILP solution

Fig. 8. ILP for inferring fence placements

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.

6.1 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. We write cost(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 a type of fence. We introduce an ILP variable t_l (in $\{0, 1\}$) for each pair (l, t).

The set potential-places is the set of such pairs that can be inserted into the program to forbid the cycles. The set actual-places is the set of such pairs that have been set to 1 by our ILP. We output this set, 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 should minimise this sum whilst satisfying the constraints.

6.2 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 Fig. 4. 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.

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 Fig. 7 (cycle 4), only a full fence can fix it (cf. Fig. 4), thus we impose $\mathsf{f}_d \geq 1$. If d is of the form poRR, as (f, h) in Fig. 7 (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 Fig. 7 limited to cycles 2, 3 and 4. Using only full fences, lightweight fences, and dependencies (i.e. ignoring control fences for now), we would generate the constraints **(i)** $\mathsf{lwf}_{(f,g)} + \mathsf{f}_{(f,g)} = 1$ for the delay (f, g) in cycle 2, (ii) $\mathsf{dp}_{(f,h)} + \mathsf{lwf}_{(f,h)} + \mathsf{f}_{(f,h)} + \mathsf{lwf}_{(g,h)} + \mathsf{f}_{(g,h)} = 1$ for the delay (f, h) in cycle 3, and **(iii)** $\mathsf{f}_{(g,h)} = 1$ for the delay (g, h) in cycle 4.

Preventing the delay (g, h) in cycle 4 requires a full fence, thus $\mathsf{f}_{(g,h)} = 1$. By the constraint **(ii)**, and since $\mathsf{f}_{(g,h)} = 1$, we derive $\mathsf{f}_{(f,g)} = 0$ and $\mathsf{lwf}_{(f,g)} = 0$. But these two equalities are not possible given the constraint **(i)**. By using inequalities, we allow several fences to live on the same edge. In fact, the constraints only ensure the soundness; the optimality is fully determined by the cost function to minimise.

Delays are in fact in po_s^+, not always in po_s: in Fig. 7, 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 Fig. 7, 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 Fig. 7, we will not impose $\mathsf{f}_{(e,g)} + \mathsf{lwf}_{(e,g)} \geq 1$ but 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 need more care, as they cannot necessarily be placed anywhere between e and g (in the formal sense of $\mathsf{between}(e, g)$): $\mathsf{dp}_{(e,f)}$ 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 Fig. 8), 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 placed after a conditional branch (e.g. bne on Power) prevent speculative reads after this branch (see Fig. 4). 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 poRR case in Fig. 8, 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 Fig. 3 has an rf connected to it via event c). In such cases, we need to use a *cumulative fence*, e.g. lwsync or sync, and not, for example, a dependency.

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 Fig. 7 (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 Fig. 3, 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 $\mathsf{potential\text{-}places}(C) \triangleq \{(l, t) \mid (t \in \{\mathsf{dp}\} \wedge l \in \mathrm{delays}(C)) \vee (t \in \mathbb{T} \backslash \{\mathsf{dp}\} \wedge l \in \bigcup_{d \in \mathrm{delays}(C)} \mathsf{between}(d)) \vee (t \in \{\mathsf{f}, \mathsf{lwf}\} \wedge l \in \mathsf{po}_s(C))\}$.

Comparison with trencher We illustrate the difference between trencher [10] and our approach using Fig. 9. 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.

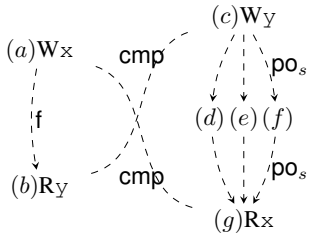


Fig. 9. Cycles sharing the edge (a, b)

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 2 locations per path, leading to 2^3 sets. Consequently, as stated in [10], 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 8 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 6 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.

	CLASSIC										FAST													
	Dek		Pet		Lam		Szy		Par		Cil		CL		Fif		Lif		Anc		Har	
LoC	50		37		72		54		96		97		111		150		152		188		179	
dfence	–	–	–	–	–	–	–	–	–	–	7.8	3	6.2	3	∼	0	∼	0	∼	0	∼	0
memorax	0.4	2	1.4	2	79.1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
musketeer	**0.0**	**5**	**0.0**	**3**	**0.0**	**8**	**0.0**	**8**	**0.0**	**3**	**0.0**	**3**	**0.0**	**1**	**0.1**	**1**	**0.0**	**1**	**0.1**	**1**	**0.6**	**4**
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	0.0	14	0.0	8	0.1	33	0.0	29	0.0	44	0.1	72
remmex	0.5	2	0.5	2	2.0	4	1.8	5	–	–	–	–	–	–	–	–	–	–	–	–	–	–
trencher	1.6	2	1.3	2	1.7	4	–	–	0.5	1	8.6	3	–	–	–	–	–	–	–	–	–	–

Fig. 10. All tools on the CLASSIC and FAST series for TSO

7 Implementation and Experiments

We implemented our new method, in addition to all the methods described in Sec. 2, in our tool musketeer, using glpk (http://www.gnu.org/software/glpk) as the ILP solver. We compare these methods to the existing tools listed in Sec. 3.

Our tool analyses C programs. dfence also handles C code, but requires some high-level specification for each program, which was not available to us. memorax works on a process-based language that is specific to the tool. offence works on a subset of assembler for x86, ARM and Power. pensieve originally handled Java, but we did not have access to it and have therefore re-implemented the method. remmex handles Promela-like programs. 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.

We now detail our experiments. CLASSIC and FAST gather examples from the literature and related work. The DEBIAN benchmarks are packages of Debian Linux 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 consists of Dekker's mutex (Dek) [14]; Peterson's mutex (Pet) [29]; Lamport's fast mutex (Lam) [21]; Szymanski's mutex (Szy) [33]; and Parker's bug (Par) [13]. We ran all tools in this series for TSO (the model common to all). For each example, Fig. 10 gives the number of fences inserted, and the time (in sec) needed. When an example is not available in the input language of a tool, we write "–". The first four tools place fences to enforce stability/robustness [5,9]; the last three to satisfy a given safety property. 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. As expected, musketeer is less precise than most tools, but outperforms all of them.

FAST gathers Cil, Cilk 5 Work Stealing Queue (WSQ) [16]; CL, Chase-Lev WSQ [11]; Fif, Michael et al.'s FIFO WSQ [26]; Lif, Michael et al.'s LIFO WSQ [26]; Anc, Michael et al.'s Anchor WSQ [26]; Har, Harris' set [12]. For each example and tool,

			TSO		Power	
	LoC	nodes	fences	time	fences	time
memcached	9944	694	3	13.9s	70	89.9s
lingot	2894	183	0	5.3s	5	5.3s
weborf	2097	73	0	0.7s	0	0.7s
timemachine	1336	129	2	0.8s	16	0.8s
see	2626	171	0	1.4s	0	1.5s
blktrace	1567	615	0	6.5s	–	timeout
ptunnel	1249	1867	2	95.0s	–	timeout
proxsmtpd	2024	10	0	0.1s	0	0.1s
ghostess	2684	1106	0	25.9s	0	25.9s
dnshistory	1516	1466	1	29.4s	9	64.9s

Fig. 11. musketeer on selected benchmarks in DEBIAN series for TSO and Power

Fig. 10 gives the number of fences inserted (under TSO) and the time needed to do so. For dfence, we used the setting of [24]: 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 [24], and write $\sim$ 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 ($0.1\,s$ vs $0.6\,s$).

DEBIAN gathers 374 executables. These are a subset of the goto-programs that have been built from packages of Debian Linux 7.1 by Michael Tautschnig. A small excerpt of our results is given in Fig. 11. The full data set, including a comparison with the methods from Sec. 2, 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 ($\sim$ 10000 LoC). Our tool needs $13.9\,s$ to place fences for TSO, and $89.9\,s$ for Power. A more meaningful measure for the hardness of an instance is the number of nodes in the aeg. For example, ptunnel has 1867 nodes and 1249 LoC. The fencing takes $95.0\,s$ for TSO, but times out for Power due to the number of cycles.

8 Conclusion

We introduced a novel method for deriving a set of fences, which we implemented 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.

Acknowledgements We thank Michael Tautschnig for the Debian binaries, Mohamed Faouzi Atig, Egor Derevenetc, Carsten Fuhs, Alexander Linden, Roland Meyer, Tyler Sorensen, Martin Vechev, Eran Yahav and our reviewers for their feedback. We thank Alexander Linden and Martin Vechev again for giving us access to their tools.

References

1. Power ISA Version 2.06 (2009)
2. Information technology – Programming languages – C. In: BS ISO/IEC 9899:2011 (2011)
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 530–536. LNCS, Springer (2013)
4. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1995)
5. Alglave, J., Maranget, L.: Stability in weak memory models. In: Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 50–66. Springer (2011)
6. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models. In: Computer Aided Verification (CAV). LNCS, vol. 6174, pp. 258–272. Springer (2010)
7. Alglave, J., Kroening, D., Lugton, J., Nimal, V., Tautschnig, M.: Soundness of data flow analyses for weak memory models. In: Programming Languages and Systems (APLAS). LNCS, vol. 7078, pp. 272–288. Springer (2011)
8. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence: A static analysis approach to automatic fence insertion. CoRR abs/1312.1411 (2013)
9. Bouajjani, A., Meyer, R., Moehlmann, E.: Deciding robustness against total store ordering. In: Automata, Languages and Programming (ICALP). LNCS, vol. 6756, pp. 428–440. Springer (2011)
10. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: European Symposium on Programming (ESOP). LNCS, vol. 7792, pp. 533–553. Springer (2013)
11. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: SPAA. pp. 21–28. ACM (2005)
12. Detlefs, D., Flood, C.H., Garthwaite, A., Martin, P.A., Shavit, N., Steele Jr., G.L.: Even better DCAS-based concurrent deques. In: Distributed Computing (DISC). LNCS, vol. 1914, pp. 59–73. Springer (2000)
13. Dice, D.: (November 2009), https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park
14. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM 8(9), 569 (1965)
15. Fang, X., Lee, J., Midkiff, S.P.: Automatic fence insertion for shared memory multiprocessing. In: International Conference on Supercomputing (ICS). pp. 285–294. ACM (2003)
16. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI. pp. 212–223. ACM (1998)
17. Krishnamurthy, A., Yelick, K.A.: Analyses and optimizations for shared address space programs. J. Par. Dist. Comp. 38(2) (1996)
18. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 111–119. IEEE (2010)
19. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI. pp. 187–198 (2011)
20. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7) (1979)
21. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1) (1987)
22. Lee, J., Padua, D.: Hiding relaxed memory consistency with a compiler. IEEE Transactions on Computers 50, 824–833 (2001)
23. Linden, A., Wolper, P.: 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, pp. 339–353. Springer (2013)

24. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. pp. 429–440. ACM (2012)
25. Marino, D., Singh, A., Millstein, T.D., Musuvathi, M., Narayanasamy, S.: A case for an SC-preserving compiler. In: PLDI. pp. 199–210. ACM (2011)
26. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. In: Principles and Practice of Parallel Programming (PPOPP). pp. 45–54. ACM (2009)
27. Norris, B., Demsky, B.: CDSchecker: checking concurrent data structures written with C/C++ atomics. In: Object Oriented Programming Systems Languages & Applications (OOPSLA). pp. 131–150 (2013)
28. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 5674, pp. 391–407. Springer (2009)
29. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
30. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. TOPLAS 10(2), 282–312 (1988)
31. Sparc Architecture Manual Version 9 (1994)
32. Sura, Z., Fang, X., Wong, C.L., Midkiff, S.P., Lee, J., Padua, D.A.: Compiler techniques for high performance sequentially consistent Java programs. In: PPOPP. pp. 2–13. ACM (2005)
33. Szymanski, B.K.: A simple solution to Lamport's concurrent programming problem with linear wait. In: ICS. pp. 621–626 (1988)
34. Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM J. Comput. 2(3), 211–216 (1973)
35. Vafeiadis, V., Zappa Nardelli, F.: Verifying fence elimination optimisations. In: Static Analysis (SAS). LNCS, vol. 6887, pp. 146–162. Springer (2011)