Partial-Coherence Abstractions for Relaxed Memory Models

Michael Kuperstein
Technion, Haifa, Israel
mkuper@cs.technion.ac.il

Martin Vechev
IBM T.J. Watson Research Center
mtvechev@us.ibm.com

Eran Yahav *
Technion, Haifa, Israel
yahave@cs.technion.ac.il

Abstract

We present an approach for automatic verification and fence inference in concurrent programs running under relaxed memory models. Verification under relaxed memory models is a hard problem. Given a finite state program and a safety specification, verifying that the program satisfies the specification under a sufficiently relaxed memory model is undecidable. For stronger models, the problem is decidable but has non-primitive recursive complexity.

In this paper, we focus on models that have store-buffer based semantics, e.g., SPARC TSO and PSO. We use abstract interpretation to provide an effective verification procedure for programs running under this type of models. Our main contribution is a family of novel *partial-coherence* abstractions, specialized for relaxed memory models, which partially preserve information required for memory coherence and consistency. We use our abstractions to automatically verify programs under relaxed memory models. In addition, when a program violates its specification but can be fixed by adding fences, our approach can automatically infer a correct fence placement that is optimal under the abstraction. We implemented our approach in a tool called BLENDER and applied it to verify and infer fences in several concurrent algorithms.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]

General Terms Algorithms, Verification

Keywords Concurrency, Synthesis, Abstract Interpretation, Relaxed Memory Models, Weak Memory Models

1. Introduction

In the early 1990s, features like out-of-order execution and multi-level caches became common in commodity CPU architectures. These features drastically improved performance in a programmer-transparent fashion: their introduction did not change the semantics of existing (sequential) programs. With the advent of symmetric multiprocessing and multi-core CPUs, preserving the illusion of in-order memory operations became more difficult. One possible approach is to keep the illusion—known as *sequential consistency* [20] in a multi-processor setting—and sacrifice performance. The other is to define architectural *relaxed memory models* (RMMs) that allow improved performance at the cost of weaker semantics.

* Deloro Fellow

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...$10.00

A relaxed memory model allows observable executions that cannot occur if instructions running on different processors are simply interleaved. As a result, a program that runs correctly on the sequentially-consistent model may violate its specification when running on a relaxed memory model. In practice, relaxed memory models are used by all major CPU designs, among them Intel x86 [33], SPARC [36] and PowerPC [16]. To enforce order between memory operations, these architectures provide special *memory fence* instructions. Informally, inserting a fence instruction prohibits certain re-orderings, thus restricting the set of relaxed executions. For example, a fence inserted between two store instructions will force these two stores to appear to execute in-order. It is the programmer's (or the compiler's) responsibility to correctly place fences.

Program Verification and Fence Inference To place fences, the programmer must first be able to know whether the program is correct given a fence placement - in other words, she needs to be able to verify the program. However, verification of concurrent programs is not an easy task even under the sequentially consistent memory model. Relaxed memory models make reasoning about program correctness, both manually and automatically, even harder, as they require reasoning about non sequentially-consistent executions. Even for finite-state programs, automatic verification under relaxed memory models is a hard problem. Given a finite state program and a safety specification, verifying that the program satisfies a specification under a sufficiently relaxed memory model (e.g., SPARC RMO) is undecidable. For somewhat stronger memory models (e.g., SPARC TSO, PSO), the problem is decidable but has non-primitive recursive complexity [2].

Even given a verification procedure, inserting fences is still non-trivial. On the one hand, since each fence incurs a heavy performance penalty, the programmer should not insert fences unless they are strictly required for correctness. On the other hand, missing a fence may lead to subtle concurrency bugs.

Store Buffers Relaxed memory models allow two basic relaxations of sequential consistency: memory operations may be re-ordered with respect to each other, and stores may be executed non-atomically across processors [1]. Some relaxations can be naturally modeled using store buffers [25], emulating the actual hardware implementation. In store-buffer based semantics, one or more FIFO queues ("store buffers") are associated with each processor. Memory writes are split into two phases: a "store" phase and a "flush" phase. The store phase adds a value into a local store buffer, and the flush phase propagates the stored value to main memory (or directly to other processors).

The basic hurdle for automatic verification under those models is that store buffers can grow without a bound, even for programs that are otherwise finite state. To enable automatic program verification and fence inference on relaxed memory models, we need a technique that can represent those buffers in a bounded way.

Existing Approaches Existing approaches either employ under-approximations such as bounded checking [5] and testing [8], or side-step the problem by focusing on a restricted class of programs. For instance, [32] considers data-race free programs, and [28] focuses on programs free from a particular ("triangular") type of data races. Bounded checking and testing are valuable, but cannot establish that the program satisfies its specification on all executions. When used for automatic fence inference, bounded techniques (e.g., [5, 18]) might miss required fences. Targeting only race-free programs simplifies the problem by allowing consideration of only sequentially-consistent executions. However, it is often unrealistic, as many concurrent programs contain "benign data races" [27]. For example, some mutual exclusion algorithms, such as Dekker's algorithm, contain benign triangular data races. Thus we cannot apply the results of [28], even if we restrict attention only to the TSO model.

In contrast to these approaches, our technique *over-approximates* possible program behaviors and is able to *verify* programs executing under RMMs. When used for fence inference, our technique is guaranteed to produce all required fences.

Our Approach We present a technique for automatic verification and fence inference in finite-state programs running on relaxed memory models. Based on abstract interpretation [9], we introduce a family of *partial-coherence* abstractions for store buffers. Our abstractions provide a bounded representation for (potentially) unbounded store buffers. We use the term *abstract memory model* to refer to a memory model that uses an abstract structure to represent store buffers. Our approach provides a range of abstractions with varying precision, enabling successive abstraction refinements of a given abstract memory model M_A.

Given a program P, a specification S and an abstract memory model M_A, the question we are trying to answer is whether $P \models_{M_A} S$, that is, whether the program satisfies the specification under the given abstract memory model. When $P \not\models_{M_A} S$, it is possible to:

- *Refine the abstraction*: refine M_A and try to find a more precise memory model $M_{A'}$ under which $P \models_{M_{A'}} S$.
- *Restrict the program*: find a program P' obtained from P by adding memory fences that restrict the permitted re-orderings during execution, such that $P' \models_{M_A} S$.

In this work, we focus on restricting the program by inserting fences, and show how using different abstract memory models affects the precision of the resulting fence placement. We focus on a family of abstractions for the TSO and PSO memory models, as those models are implemented in common hardware (e.g., Intel x86 [29], SPARC) and have simple concrete operational semantics.

Partial-Coherence Abstractions The challenge for abstractions of store buffers is to provide a bounded representation that (partially) preserves the following three key properties (described in more detail in Section 2):

- Intra-process memory coherence: a process should only see its own most recently written value to a variable.
- Inter-process memory coherence: a process should observe values written by another process in the order they were written.
- Fence semantics: a fence executed by a process writes to memory the most recent value written by the process.

The main idea behind our abstractions is to preserve only a limited amount of order inherent in a store buffer. In particular, our abstract buffer representation preserves information about: i) the most recent store to a buffer, and ii) the order between a bounded number of the oldest stores in the buffer. While inter-process coherence is only partially preserved, we show this choice is particularly effective for verifying concurrent algorithms running on relaxed memory models (see Section 4 for details).

```
Process 0:                    Process 1:

while(true)                   while(true)
{                             {
 store ent0 = true;            store ent1 = true;
 store turn = 1;               store turn = 0;
 do                            do
 {                             {
  load e = ent1;                load e = ent0;
  load t = turn;                load t = turn;
 }                             }
 while(e==true && t==1);       while(e==true && t==0);
 //Critical Section            //Critical Section
 store ent0 = false;           store ent1 = false;
}                             }
```

Figure 1. Peterson's Algorithm with explicit memory operations

```
Process 0:                    Process 1:

while(true)                   while(true)
{                             {
 store ent0 = true;            store ent1 = true;
 fence;                        fence;
 store turn = 1;               store turn = 0;
 fence;                        fence;
 do                            do
 {                             {
  load e = ent1;                load e = ent0;
  load t = turn;                load t = turn;
 }                             }
 while(e==true && t==1);       while(e==true && t==0);
 //Critical Section            //Critical Section
 store ent0 = false;           store ent1 = false;
}                             }
```

Figure 2. Peterson's Algorithm with fences that guarantee mutual exclusion under the PSO memory model. Fences were automatically inferred by our approach.

1.1 Main Contributions

The main contributions of this paper are as follows:

- We describe a family of parametric abstractions that enable automatic verification of safety properties for programs under relaxed memory models.
- When a program violates its specification but can be fixed by adding fences, our approach can automatically infer a correct fence placement that is optimal under the given abstraction.
- We have implemented our approach in a tool called BLENDER and applied it for verification and fence inference of several challenging concurrent algorithms.

2. Overview

2.1 Motivating Example - Peterson's Algorithm

Fig. 1 shows the code of Peterson's mutual exclusion algorithm [31]. In this algorithm, two processes repeatedly enter and exit a critical section. We would like to show that the algorithm satisfies the *mutual exclusion* property: it is impossible for both processes to be in the critical section simultaneously. Indeed, Peterson's algorithm satisfies mutual exclusion under a sequentially-consistent (SC) memory model. Unfortunately, under relaxed memory models, such as "Partial Store Order" (PSO), the algorithm *does not* satisfy the property. To see why, we first give a brief explanation of the PSO memory model.

The Partial Store Order (PSO) Memory Model PSO is one of three memory consistency models defined for the SPARC architecture [36]. In PSO, a store to some memory location l may become

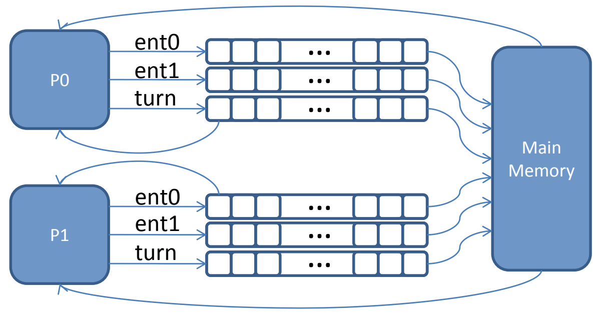


Figure 3. Store buffers for Peterson's algorithm of Fig. 1 under the PSO memory model. Note that buffers can grow without a bound.

visible to other processes only after the storing process executes later loads and stores to different memory locations.

The PSO model can be formalized operationally by associating with each processor a set of FIFO queues (store buffers), one for each variable, as shown in Fig. 3. The informal semantics of store buffers for PSO can be summarized as follows:

- Store buffering: A store issued by process p_i to variable x is written into the store buffer associated with (p_i, x).
- Store forwarding: A load by p_j from y is performed from its local store buffer (associated with (p_j, y)) if it is not empty, or from the global memory otherwise.
- Flushing: The oldest value stored in the buffer may be written to the global memory and removed from the buffer at non-deterministic points in the execution.

The Problem: Delayed Stores Under the PSO model, the following execution of Peterson's algorithm is possible:

- p_0 runs alone until line 11, however the store to `ent0` in line 3 is written only to the buffer but not flushed.
- p_1 runs. Since the store to `ent0` is delayed, it is not visible to p_1. p_1 enters the critical section, and mutual exclusion is violated.

Peterson's algorithm relies on *ordering* of loads and stores for synchronization. It requires p_0's store to `ent0` to be visible to p_1 before p_0 loads `ent1`, and symmetrically on p_1's store to `ent1` to be visible to p_0 before p_1 loads `ent0`. When the underlying memory model does not preserve this order, Peterson's algorithm, as it appears in Fig. 1, does not satisfy mutual exclusion.

Restoring Order with Fences To allow programmer control over ordering in relaxed memory models, processors provide special memory fence instructions. Intuitively, the semantics of a fence are that memory operations issued before the fence must take global effect before memory operations after the fence may execute. In general, there are different kinds of fences (e.g., store-load, store-store) that impose order between different types of operations. A store-load fence executed by a processor forces all stores issued by that processor to complete before any new loads by the same processor start. In this paper we assume the model provides the strongest type of fence (a "full memory barrier") that restricts reordering of any memory operations. In Fig. 2 the fences in lines 4 and 6 prevent the erroneous execution above (and other possible related bugs) by forcing the stores in lines 3 and 5 to take global effect before the storing process can advance. Unfortunately, fence instructions are very costly in terms of CPU cycles. Thus, we wish to place fences only when they are required for correctness.

Efficient Fence Placement The programmer's challenge is, then, in finding a fence placement that permits as much re-ordering as possible but does not allow the specification to be violated. To find an efficient placement of fences, we need to observe what re-orderings lead to violation of the specification, and find a minimal placement (often, there are multiple non-comparable solutions) that prevents these re-orderings. When the program is finite-state, we can enumerate all reachable program states, identify error states and find fences that prevent execution from reaching those states (c.f [18]). Unfortunately, Peterson's algorithm without fences running on PSO has an infinite state-space. The length of the store buffers generated by the program is not bounded: running p_0 alone for t iterations of the outer loop without flushing will generate a buffer of length $2t$ for the `ent0` variable.

2.2 Abstraction

To handle programs that have an unbounded state-space, we introduce a family of parametric abstractions that provide a conservative bounded representation. Our abstractions induce a hierarchy of (abstract) memory models with varying degrees of consistency. Before describing the abstraction, we note that concrete PSO semantics preserve the following 3 properties.

1. Intra-process coherence: If a process stores several values to shared variable x, and then performs a load from x, it should not see any value it has itself stored except the most recent one.
2. Inter-process coherence: A process p_i should not observe values written to shared variable x by process p_j in an order different from the order in which they were written.
3. Fence semantics: If a process p_i executes a fence when its buffer for variable x is non-empty, the value of x visible to other processes immediately after the fence should be the most recent value p_i wrote.

The properties above are phrased in terms of PSO semantics (store buffer per variable), but it is easy to formulate similar properties for other memory models. For example, for TSO, the only change is that inter-process coherence is global and not per variable. In that case, the desired property may be called inter-process *consistency*.

Partial-Coherence Abstraction The challenge in designing an abstraction for store-buffer based memory models lies in preserving properties 1-3 (to the greatest possible extent) using a bounded representation of each buffer. To preserve intra-process coherence, our abstractions maintain recency information per variable. To preserve inter-process coherence, our abstractions preserve order between stores up to some constant bound (a parameter of our abstraction), and treat the remaining stores as an unordered set. While property 2 is not fully preserved, this *partial coherence* is often sufficient in practice. The intuition is that if a process stores many (possibly different) values to the same shared variable without an intervening fence, the order in which they become visible is not important for the correctness of the algorithm.

Fig. 4 shows a schematic view of a partial-coherence abstraction of PSO store buffers for the Peterson algorithm of Fig. 1. In this abstraction, a store buffer is represented by treating items after a bounded head (of length k) of the buffer as a set, and additionally recording the most recently stored value for each buffer.

In Section 3, we give a formal concrete semantics for the PSO memory model and in Section 4 present our abstract semantics. Using this abstraction with the fence inference algorithm of Section 5, our approach automatically infers the fences shown in Fig. 2. In Section 6, we show that we can use different parameters to achieve more scalable abstractions and still get reasonable results. We also show that there is a tradeoff between the precision of the abstraction and the quality of the inferred fences. Finer abstractions lead to successful inference with fewer fences, while restricting the program by adding fences enables verification with a coarser abstraction. In particular, our partially disjunctive abstraction (see Section 4.3) produces non-trivial fence placements for programs for which the fully disjunctive abstraction leads to state-space explosion.

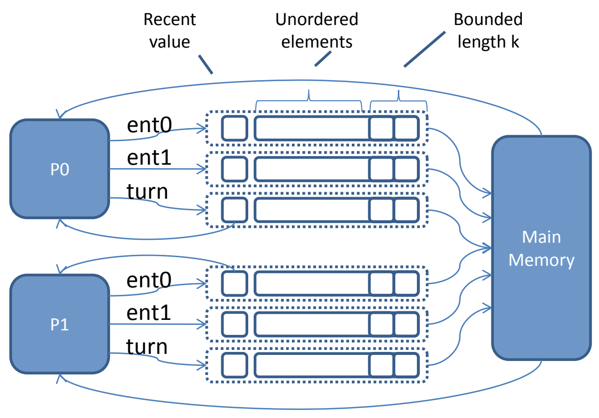


Figure 4. A partial-coherence abstraction of PSO store buffers for the Peterson algorithm of Fig. 1. In this abstraction, a store buffer is given a bounded representation by representing items after a bounded head of the buffer as a set, and recording the recently stored value for each buffer.

3. Operational Semantics for Relaxed Memory Models

In this section, we present an operational semantics for the PSO memory model. It is easy to give similar semantics for other conceptually close models such as TSO, NTSO and NPSO [24].

3.1 Preliminaries

Sequence Notation Given a finite domain D, we use $Seq_n(D)$ to denote the set of all sequences of length n over D, $Seq_{\leq n}(D)$ to denote the set of all sequences shorter than or equal in length to n over D, $Seq(D)$ to denote the set of all finite sequences over D, $|w|$ to denote the length of a sequence w and ϵ to denote an empty sequence. We denote the concatenation of two sequences w_1, w_2 by $w_1 \cdot w_2$. For $k > |w|$, we define $head(w, k)$ as the subsequence consisting of the first k items in w and $tail(w, k)$ as the subsequence consisting of the last k items in w. For $0 < k \leq |w|$ we define $head(w, k) = tail(w, k) = w$ and for $k \leq 0$, $head(w, k) = tail(w, k) = \epsilon$. We define $last(w)$ to be the only element in $tail(w, 1)$, or $\perp$ if $tail(w, 1) = \epsilon$. We define $Set(w)$ to be the set of elements in the sequence w. Finally, we define $UTail(w, k)$ as $Set(tail(w, |w| - k))$ — the set of all but the first k elements of w.

Program Syntax We consider programs written in a simple assembly-like programming language with the operations load, store, branch, CAS (compare and swap) and sequential and parallel composition. Our language also contains a full fence operation. We assume that instructions in our programs are labeled, and the labels used in the code of process are unique. We denote the set of program labels by *Labs*.

Program Semantics A transition system for a program P under a memory model M is a tuple $\langle \sigma_0, \Sigma, T \rangle$, where Σ is a set of states, $\sigma_0 \in \Sigma$ is the initial state of P, and T is a set of transitions $\sigma \xrightarrow{t} \sigma'$. A transition $\sigma \xrightarrow{t} \sigma'$ is in T if $\sigma, \sigma' \in \Sigma$, and execution from state σ according to the semantics of M can result in state σ'. A trace π of the program is a (possibly infinite) sequence of transitions $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots$, where for all i, $\sigma_i \longrightarrow \sigma_{i+1} \in T$. In all of our semantics, a single transition will correspond to action taken by a single process. Thus we will associate a transition t with that process, and denote the associated process $proc(t)$. A transition t_p is *enabled* for process p in state σ if $p = proc(t_p)$ and there exists some σ_2 such that $\sigma \xrightarrow{t_p} \sigma_2 \in T$.

Throughout the paper we present the semantics in a standard operational style as a set of inference rules. To simplify presentation, when updating mappings, we use $M'(x) = v$ as a shorthand for $M' = M[x \mapsto v]$. Components not updated in the inference rule are assumed to be left unchanged.

3.2 Store Buffers

In our memory model semantics we follow [2, 7, 25] and assume that processes execute their programs sequentially, and any violations of sequential consistency happen within the memory subsystem. This is in contrast to other formulations that combine the memory and processor effects (e.g., [30, 35, 41]).

Our formulation is based on store-buffers, and our concrete semantics uses the following semantic domains:

- $G^\natural \in SVar$ where $SVar = Shared \rightarrow D$. Valuation of shared variables from the domain D.
- $L^\natural \in Env$ where $Env = PID \rightarrow (Local \rightarrow D)$. Valuation of local variables for each process.
- $pc^\natural \in PC$ where $PC = PID \rightarrow Labs$. Program counters.
- $B^\natural \in SB$ where SB differs between different memory models, and is intentionally left unspecified at this stage. A representation of the store buffers.

Here D represents the domain from which the variables in the program take values.

A state $\sigma = \langle G^\natural_\sigma, L^\natural_\sigma, pc^\natural_\sigma, B^\natural_\sigma \rangle \in C^\natural$ is a tuple where $C^\natural = SVar \times Env \times PC \times SB$. We use $next(pc(p))$ to mean the instruction following $pc(p)$ in the program code. Furthermore, we will omit the p when the referenced process is clear from the context.

3.3 Partial Store Order (PSO) Model

Concrete Semantics For PSO, a separate FIFO store buffer is maintained for every (process, variable) pair. That is, formally, $SB = PID \rightarrow (Shared \rightarrow Seq(D))$

Semantics 1 Operational semantics defining transition from $\langle G, L, pc, B \rangle$ to $\langle G', L', pc', B' \rangle$ under PSO.

$$\frac{stmt(pc) = load\ x, r \quad B(x) = \epsilon \quad G(x) = v}{L'(r) = v \quad pc' = next(pc)} \ (\text{LOAD-G})$$

$$\frac{stmt(pc) = load\ x, r \quad B(x) = b \cdot v}{L'(r) = v \quad pc' = next(pc)} \ (\text{LOAD-B})$$

$$\frac{stmt(pc) = store\ r, x \quad B(x) = b \quad L(r) = v}{B'(x) = b \cdot v \quad pc' = next(pc)} \ (\text{STORE})$$

$$\frac{B(x) = v \cdot b}{B'(x) = b \quad G'(x) = v} \ (\text{FLUSH})$$

$$\frac{stmt(pc) = fence \quad \forall x. B(x) = \epsilon}{pc' = next(pc)} \ (\text{FENCE})$$

$$\frac{\begin{array}{c} stmt(pc) = cas\ x, r, s, q \\ G(x) = L(r) \quad L(s) = v \quad B(x) = \epsilon \end{array}}{G'(x) = v \quad L'(q) = true \quad pc' = next(pc)} \ (\text{CAS-T})$$

$$\frac{stmt(pc) = cas\ x, r, s, q \quad G(x) \neq L(r) \quad B(x) = \epsilon}{L'(q) = false \quad pc' = next(pc)} \ (\text{CAS-F})$$

Sem. 1 shows the concrete operational semantics of the PSO model. Each inference rule applies only to a single process. Thus, the p parameter is consistently omitted in all inference rules presented. However it is always implicitly existentially quantified. For example, the premise of the LOAD-G rule should be read as

$$\exists p.stmt(pc(p)) = load\,x, r \wedge B(p)(x) = \epsilon \wedge G(x) = v$$

The semantics show the role played by the store buffer for storing and loading values to/from main memory (STORE, LOAD-G, LOAD-B, FLUSH). The FENCE and CAS rules have memory fence semantics. These two rules are enabled only when the buffer of the executing process is empty. This means that when a process encounters, e.g., a fence instruction, it cannot continue execution until all of the buffers are flushed. For simplicity we omit the semantics of instructions that do not access shared memory (register operations, branches) and leave expression evaluation implicit. That is, $L(r)$ is extended to the evaluation of complex expressions r. Such a complex expression may only depend on local variables — expression evaluation may not cause a memory access.

The premise of all rules except FLUSH depends on the program counter of the process. They are enabled only if $pc(p)$ points to an instruction of a specific type. The FLUSH rule, on the other hand, is always enabled for a given buffer $B(p)(x)$ if that buffer is not empty. This captures the fact that flushes can be performed non-deterministically at any stage of program execution.

3.4 Total Store Order (TSO) Model

The TSO concrete state differs from the PSO concrete state only in the definition of the store buffer. For TSO, there is only a single, buffer for all variables of a process. That is, $SB = PID \rightarrow Seq(Shared \times D)$. The semantics must also be updated to take the difference into account. The flavor of the required changes can be seen in the TSO version of the LOAD-G rule in Sem. 2. Note that as the difference between PSO and TSO lies purely in the grouping of shared variables into store buffers, we can treat them as special cases of the same general model.

Semantics 2 LOAD-G rule for concrete TSO

$$\frac{stmt(pc) = load\,x, r \qquad \forall(y,d) \in B.y \neq x \qquad G(x) = v}{L'(r) = v \qquad pc' = next(pc)} \; (\text{LOAD-G})$$

4. Partial-Coherence Abstractions

In this section, we present a family of abstract memory models that abstract the concrete semantics of Section 3. The presentation focuses on abstractions of the SPARC PSO model, but the adaptation to TSO is straight-forward. The main idea behind our *partial-coherence* abstractions is to vary how much of the order between memory operations we preserve. The ability to vary the precision is useful as different algorithms can be verified with different levels of precision and cost. When the abstraction is used for fence inference, Section 6 shows that there exists a trade-off between the precision of the analysis (which affects the state-space size) and the quality of inferred fences.

4.1 Abstract Domain

The abstract domain is designed to represent store buffers in a bounded way by losing order information between items past a certain bound. To achieve this goal, we represent a concrete buffer $B^\natural$ by a tuple $\langle l, S, H\rangle$. The $l \in D$ element records the latest (most recent) value that was written into the buffer. $H \in Seq_{\leq k}(D)$ records the k oldest values in the buffer (in the original order) if those are known. $S \subseteq D$ records a set of values that were written into the buffer, abstracting away the order between them, as well as the number of times each elements appears in the buffer. Formally, we define, for a buffer $B^\natural$:

$$\beta_B(B^\natural) = \langle last(B^\natural), UTail(B^\natural, k), head(B^\natural, k)\rangle$$

An abstract state σ is a tuple $\langle G, L, pc, B\rangle$ where G, L and pc are defined as in the concrete semantics. B maps a (process, shared variable) pair to the tuple $\langle l, S, H\rangle$ defined earlier. To simplify notation, we will use short-hands such as $l_p(x)$ to represent the l element of $B(p)(x)$. As in the concrete semantics, we will often omit the p. We denote by A the set of all abstract states.

To define the abstract domain, we define several order relations. The order $\sqsubseteq_b$ is defined on $\langle l, S, H\rangle$ tuples:

$$\langle l_1, S_1, H_1\rangle \sqsubseteq_b \langle l_2, S_2, H_2\rangle$$

if $l_1 = l_2$ and:

$$\exists t.H_2 = head(H_1, t) \wedge S_2 = S_1 \cup UTail(H_1, t)$$

Intuitively, $\langle l_2, S_2, H_2\rangle$ is produced from $\langle l_1, S_1, H_1\rangle$ by removing part of the tail end of H_1 and adding all the removed elements into the set. We then use $\sqsubseteq_b$ to define a partial order $\sqsubseteq_s$ on abstract states $\sigma_1 = \langle G_1, L_1, pc_1, B_1\rangle, \sigma_2 = \langle G_2, L_2, pc_2, B_2\rangle$: $\sigma_1 \sqsubseteq_s \sigma_2$ if the two states coincide on G, L, pc and:

$$\forall p, x.B_1(p)(x) \sqsubseteq_b B_2(p)(x)$$

Finally, we define our abstract domain $\mathbb{A} \subseteq 2^A$ as the set of all antichains of A. That is:

$$\mathbb{A} = \{\Sigma \subseteq A \mid \forall \sigma, \rho \in \Sigma.\sigma \neq \rho \Rightarrow \sigma \not\sqsubseteq_s \rho\}$$

The order relation $\sqsubseteq\colon \mathbb{A} \times \mathbb{A}$ is defined as:

$$\Sigma_1 \sqsubseteq \Sigma_2 \texttt{ iff } \forall \sigma_1 \in \Sigma_1.\exists \sigma_2 \in \Sigma_2.\sigma_1 \sqsubseteq_s \sigma_2$$

The join operator over $\mathbb{A}$, implied by the above order, is:

$$\Sigma_1 \sqcup \Sigma_2 = \{\sigma \in \Sigma_1 \cup \Sigma_2 \mid \forall \rho \in (\Sigma_1 \cup \Sigma_2).\sigma \neq \rho \Rightarrow \sigma \not\sqsubseteq \rho\}$$

We define the abstraction function $\alpha\colon 2^{C^\natural} \rightarrow \mathbb{A}$ using an extraction function $\beta\colon C^\natural \rightarrow A$:

$$\alpha(\Sigma) = \bigsqcup_{\sigma \in \Sigma}\{\beta(\sigma)\}$$

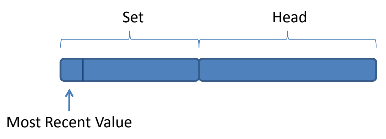
$$\beta(\sigma) = \langle G^\natural_\sigma, L^\natural_\sigma, pc^\natural_\sigma, \hat{\beta_B}\rangle$$

$$\hat{\beta_B} = \lambda p, x.\beta_B(B^\natural_\sigma(p)(x))$$

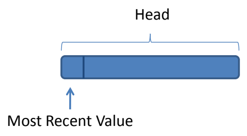
The intuition behind this abstraction is shown in Fig. 3. That figure can, however, be slightly misleading: it is important to note the entire concrete buffer is covered by the concatenation of values from S to H. Specifically, l does not need to be concatenated to the end. Thus if $S = \emptyset$, then l is always equal to the last element of H and is in fact redundant. If $S \neq \emptyset$, then $l \in S$ is an invariant. A more precise representation of the abstraction is given in Fig. 5. Fig. 5(a) shows the case in which the concrete buffer is longer than k and Fig. 5(b) shows the case in which the concrete buffer is of length at most k.

The Importance of Recency Our abstraction uses l to record the most recent value stored for a variable. This is motivated by the need to preserve the *intra-process coherence* requirement that a process storing several values to a shared variable x, and then performing a load from x, should not see any value it has itself stored except the most recent one. This is a very basic property and abstractions that do not preserve this information will fail to verify many reasonable programs.

Partial Inter-Process Coherence The abstract domain only partially preserves the *inter-process coherence* requirement. For example, suppose processor p stores the values a and then b to the variable x. The resulting concrete buffer $B^\natural_p(x)$ is "ab". Taking $k = 0$, the abstract buffer is $B = \beta_B(\text{"}ab\text{"}) = \langle a, \{a, b\}, \epsilon\rangle$. Note that, for example, $\beta_B(\text{"}ab\text{"}) = \beta_B(\text{"}aba\text{"})$. So given the abstract buffer B we must allow a different process q to observe the values being written in the opposite order. Worse, since, for example,



(a) The length of the buffer is higher than k



(b) The length of the buffer is at most k

Figure 5. Abstraction of a single buffer

$\beta_B("ab") = \beta_B("abab")$ it is possible for process q to observe the stores in the opposite order, and for a third process r to observe them in the original order.

This behavior occurs only if a process performs more than k stores to the same memory location without an intervening fence or CAS. As long as at most k stores to a memory location are performed without an intervening fence, the abstract domain is fully precise.

Abstract Domain Design One of our observations is that in a correct program a process rarely performs an unbounded number of stores to a memory location without a fence. When a process does perform a large number of stores to a memory location without a fence, it means that the order in which these stores are performed is not important for program correctness. Following this observation, our abstraction is designed to: (i) preserve the order between a small number of stores to the same memory location; (ii) abstract away the order in long sequences of stores but preserve the stored values, such that values cannot appear "out of thin air".

Given that programmers do not normally think in terms of unordered stores, we expect many correct programs to only utilize very short buffers. This is validated by our results (c.f Section 6) — correct versions of the benchmark algorithms could be verified using $k = 1$. However, there are examples of correct programs where buffers may become very long. One such example is the Sieve of Eratosthenes implementation in [3], which only requires that values do not appear out of thin air.

Furthermore, we wish our verification procedure to remain sound for arbitrary programs. This is impossible using bounded buffers, as it is trivial to construct an incorrect program which would appear to be correct using bounded buffers of some given length k. Similarly, when the abstraction is used for fence inference, we wish to always infer a correct placement. This is impossible if we simply bound the buffer length.

Domain of Variables Throughout this paper we do not specify the domain D from which local and shared variables take values. However, the domain is in fact critically important to the effectiveness of the abstraction. Let d be the number of values a shared variable may potentially take during the execution of a program. Then the number of possible H values for that variable is d^k, and the number of possible S values is 2^d. Since in the worst case, $d = |D|$, we may expect an increase in state-space size (w.r.t state-space under SC) that is exponential in $|D|$. The examples we used for our experiments do not suffer from this problem, as the variables only take a small number of values in any given execution. However, for different programs this may become a real issue. Note, however, that our abstraction can be trivially composed with value abstractions (e.g., sign abstraction, parity abstraction, interval abstraction [10]). Instead of using the concrete domain D, we can replace it with an abstract value domain $D^\sharp$. We can then store those abstract values in the buffer and perform local operations on values according to the semantics dictated by $D^\sharp$, retaining a sound abstraction.

4.2 Abstract Semantics

Sem. 3 shows the abstract semantics with partial coherence parameterized by k. In the figure, we use the shorthand $emp(x) \overset{\text{def}}{=} H(x) = \epsilon \wedge S(x) = \emptyset$.

Loading Values and Recency In the concrete semantics, a process may load the latest value it wrote by reading its own store buffer. Correspondingly, in the abstract semantics, the rule `LOAD-B` reads the most recent value recorded in l. Had we not recorded the most recent value l that a process wrote, a process p that performs a load when $S_p(x) \neq \emptyset$ would have to conservatively explore all possible values in the set $S_p(x)$. The rule `LOAD-G` is similar to the concrete semantics: when the buffer is known to be empty, the value is loaded from global store.

Storing Values In the abstract semantics, `store` is split into two cases `STORE-H` and `STORE-S`, based on whether the size of the buffer $H(x)$ has reached the bound k. As long as $|H(x)| < k$ and $S(x) = \emptyset$, the contents of the buffer are known precisely. Thus, similarly to the concrete semantics, the effect of a store follows `STORE-H`, adding the value to the tail of the buffer $H(x)$ and updating the most recent value $l(x)$. When $|H(x)| = k$, the size of the buffer H has been exceeded and no more values can be stored in H. Therefore, the new value is stored in the (unordered) set of values $S(x)$ (as shown in the rule `STORE-S`) and the most recent value $l(x)$ is updated accordingly. When $S(x) \neq \emptyset$ we have lost the information on the precise number of elements on the buffer, and thus are also forced to keep updating the set.

Flushing Values In the abstract semantics, `flush` is split into three cases: `FLUSH-H`, `FLUSH-SN` and `FLUSH-SD`. When we have $H(x) \neq \epsilon$ then `FLUSH-H` behaves as the `FLUSH` rule in the concrete semantics: it selects the oldest element in $H(x)$, writes it to $G(x)$ and updates $H(x)$. However, when $H(x) = \epsilon$ and $S(x) \neq \emptyset$, any of the values in $S(x)$ become possible candidates for flushing (since $S(x)$ is unordered, we do not know which value is the oldest one). The rules `FLUSH-SD` (*flush from set, destructive*) and `FLUSH-SN` (*flush from set, non-destructive*) then only differ on whether the value selected to be flushed is removed from $S(x)$ or kept in it. This is required since we do not know how many times every value appears in the buffer. Thus, in the concrete domain, `FLUSH-SD` of a value v represents a flush of the last occurrence of v in the buffer. In contrast, `FLUSH-SN` represents the situation in which more instances of v remain.

We improve the precision of the analysis by disabling the `FLUSH-SD` rule when we know that the resulting abstract states do not represent any possible concrete states and will only introduce imprecision. In particular, if $v = l(x)$ and $S(x) \neq \{v\}$, `FLUSH-SD` need not fire. If we apply the concrete `FLUSH` rule to any concretization of such a state, the value v will stay in the (concrete) buffer, while if we flush v from the abstract state using `FLUSH-SD`, it will remove v from the abstract buffer, leading to abstract states that could not arise in the concrete semantics.

Example: Motivating Recency and Order Next, we illustrate via an example why maintaining recency and order is important for verification and inference. Consider a naive set abstraction for the store buffers, and a version of Peterson's algorithm with fences shown in Fig. 2 (in Section 2). Under standard concrete semantics of PSO,

Semantics 3 Abstract operational semantics defining transition from $\langle G, L, pc, B\rangle$ to $\langle G', L', pc', B\rangle$

$$\frac{stmt(pc) = load\, x, r \qquad emp(x) \qquad v = G(x)}{L'(r) = v \qquad pc' = next(pc)} \; \text{(LOAD-G)}$$

$$\frac{stmt(pc) = load\, x, r \qquad \neg emp(x) \qquad v = l(x)}{L'(r) = v \qquad pc' = next(pc)} \; \text{(LOAD-B)}$$

$$\frac{\begin{array}{c} stmt(pc) = store\, r, x \\ S(x) = \emptyset \qquad H(x) = h \qquad |h| < k \qquad L(r) = v \end{array}}{H'(x) = h \cdot v \qquad l'(x) = v \qquad pc' = next(pc)} \; \text{(STORE-H)}$$

$$\frac{\begin{array}{c} stmt(pc) = store\, r, x \\ S(x) = s \qquad s \neq \emptyset \vee |H(x)| = k \qquad L(r) = v \end{array}}{S'(x) = s \cup \{v\} \qquad l'(x) = v \qquad pc' = next(pc)} \; \text{(STORE-S)}$$

$$\frac{H(x) = v \cdot h}{H'(x) = h \qquad G'(x) = v} \; \text{(FLUSH-H)}$$

$$\frac{H(x) = \epsilon \qquad v \in S(x)}{G'(x) = v} \; \text{(FLUSH-SN)}$$

$$\frac{H(x) = \epsilon \qquad v \in S(x) \qquad \neg(v = l(x) \wedge S(x) \neq \{v\})}{G'(x) = v \qquad S'(x) = S(x) \setminus v} \; \text{(FLUSH-SD)}$$

$$\frac{stmt(pc) = fence \qquad \forall x. emp(x)}{pc' = next(pc)} \; \text{(FENCE)}$$

$$\frac{\begin{array}{c} stmt(pc) = cas\, x, r, s, q \\ emp(x) \qquad L(r) = G(x) \qquad L(s) = v \end{array}}{G'(x) = v \qquad L'(q) = true \qquad pc' = next(pc)} \; \text{(CAS-T)}$$

$$\frac{stmt(pc) = cas\, x, r, s, q \qquad emp(x) \qquad L(r) \neq G(x)}{L'(q) = false \qquad pc' = next(pc)} \; \text{(CAS-F)}$$

those fences guarantee that it is impossible for both processes to be concurrently executing line 13. Let us consider an abstract memory model where order and recency are not maintained, that is, we only maintain $S_p(x)$ but without maintaining $l_p(x)$ and $H_p(x)$. Then, we cannot show that the algorithm is correct. Consider the following execution:

1. Initially both processes start with empty buffers, and $ent0 = ent1 = turn = 0$.
2. Process 0 runs through one iteration of the outer loop (executes lines 1-14 inclusively), without performing a flush after line 14.
3. Process 0 then tries to enter the critical section again and executes lines 1-3 inclusively. At this stage, $S_{p_0}(ent0) = \{true, false\}$.
4. Two flush actions are performed on $S_{p_0}(ent0)$, first flushing $true$ and then $false$. At this point $G(ent0) = false$.
5. Process 0 completes entering the critical section.
6. Process 1 loads ent0 from global store and since ent0 is $false$ process 1 also enters the critical section.

The above example would not have been possible had we kept either: i) ordering information via $H_p(ent0)$ for at least two values (i.e., $k = 2$) or ii) recency information via $l_p(ent0)$. In the first case, the order in which $\{true, false\}$ are flushed would have been consistent with the order in which the values were written: we would have first flushed $false$ and then $true$. In the second case, the fence in line 4 would have forced fully flushing $S_{p_0}(ent0)$, resulting in writing out the most recent value (i.e., $G(ent0) = true$). While in this case we could have used either $l_p(ent0)$ or $H_p(ent0)$ with $k = 2$, in other examples both of these refinements with respect to a set are required.

4.3 A Partially Disjunctive Abstraction for Store Buffers

The abstraction of Section 4.1 distinguishes two abstract buffers $B_1 = \langle l_1, S_1, H_1\rangle, B_2 = \langle l_2, S_2, H_2\rangle$ even when they differ only on the contents of their unordered sets $S_1 \neq S_2$. This leads to distinctions between abstract states that are often more precise than necessary. We observe that a more efficient abstraction can be obtained without a significant sacrifice in precision by merging such states. In Section 6, we show that combining such states leads to a more scalable abstraction, while keeping a sufficient level of precision. The one distinction that we do wish to preserve regarding the S component is the difference between an empty set and a non-empty set, as many of the rules in Sem. 3 distinguish between these two cases. To capture this, we change the definition of $\sqsubseteq_b$ as follows: The order $\sqsubseteq_b^\tau$ is defined on $\langle l, S, H\rangle$ tuples.

$$\langle l_1, S_1, H_1\rangle \sqsubseteq_b^\tau \langle l_2, S_2, H_2\rangle$$

if $l_1 = l_2$ and one of the two following conditions holds:

$$(1)\; H_1 = H_2 = \epsilon \wedge S_1 = S_2 = \emptyset$$

$$(2)\; S_1 \neq \emptyset \wedge \exists t. H_2 = head(H_1, t) \wedge S_2 \supseteq S_1 \cup UTail(H_1, t)$$

The orders $\sqsubseteq_s^\tau$ and $\sqsubseteq^\tau$ are then defined exactly as before but with respect to $\sqsubseteq_b^\tau$ instead of $\sqsubseteq_b$. α^τ is also defined as before, but using $\bigsqcup^\tau$ instead of $\bigsqcup$. Note that this small change in the formalism drastically changes the intuitive meaning of the set S. Let $B = \langle l, S, \epsilon\rangle$ be an abstract buffer, and $B^\natural$ a concrete buffer such that $\beta_B(B^\natural) \sqsubseteq_b^\tau B$. In the fully-disjunctive abstraction, this implies that $Set(B^\natural) = S$. This means that a value v was in S if and only if it appeared at least once in $B^\natural$. In the partially-disjunctive abstraction, this is no longer true. Consider the concrete buffer "a". Assuming $k = 0$, $\beta_B(\text{"}a\text{"}) = \langle a, \{a\}, \epsilon\rangle \sqsubseteq_b \langle a, \{a, b\}, \epsilon\rangle$. If a value appears at least once in $B^\natural$ then it is necessarily in S, but the converse does not hold.

The new abstraction also implies a change to the abstract transformer. In the fully disjunctive abstraction, flushes from S were split into two cases: `FLUSH-SD` to represent flushing the last instance of a value from the buffer, and `FLUSH-SN` to represent an instance that is not the last one. The case split for the partially disjunctive abstraction is slightly different. The new flush semantics are shown in Sem. 4. The rule `FLUSH-NE` covers the case in which a flush leaves S non-empty, while `FLUSH-E` represents flushing the only remaining element of the concretization of the abstract buffer. Note that it's possible for both types of flush rules to be enabled for the same buffer.

Semantics 4 Partially-disjunctive flush semantics

$$\frac{H(x) = \epsilon \qquad v \in S(x)}{G'(x) = v} \; \text{(FLUSH-NE)}$$

$$\frac{H(x) = \epsilon \qquad S(x) \neq \emptyset}{G'(x) = l(x) \qquad S'(x) = \emptyset} \; \text{(FLUSH-E)}$$

5. Fence Inference

In this section, we introduce a new technique for inferring memory fences under store-buffer based abstract memory models. For our algorithm, we follow the same general recipe as outlined in [40]: (i) Construct (a possibly abstract) transition system and find the reachable error states. (ii) Construct a boolean formula that describes how traces leading to those error states can be avoided. (iii) Implement satisfying assignments of the formula using syntactic constructs.

The main challenge in implementing this general recipe is in defining which transitions may be avoided, and how they can be avoided syntactically. Next, we present the definitions that are appropriate for each of our semantics: concrete, fully-disjunctive and partially-disjunctive. We also show an example demonstrating the interplay between the precision of the abstraction and the quality of inferred fences.

5.1 Recoverability of Sequential Consistency

Before we introduce the details of the inference algorithm, we note that the problem always has a trivial (inefficient) solution under concrete memory models. However, for some abstract memory models, the problem no longer has a solution. We would like to restrict attention to abstractions in which the existence of a solution is guaranteed. Consider a program P that satisfies its specification S under the sequentially consistent memory model, $P \models_{SC} S$, but violates it under a weaker memory model M, $P \not\models_M S$. We say that M is *SC-Recoverable (SCR)* when for any such P there exists a program P' obtained from P by adding fences such that $P' \models_M S$. For SC-Recoverable memory models, when $P \models_{SC} S$, the trivial solution in which fences are added after every memory store in P always exists. This property might seem trivial, however it is easy to design seemingly reasonable abstract models for which it does not hold. For instance, as demonstrated in Section 6, a partial-coherence abstraction with recency and $k = 0$ does not satisfy the SCR property.

For the partial coherence abstractions of Section 4, $k \geq 1$ guarantees SC-Recoverability. If we place a fence immediately after every store instruction, then i) $|H|$ can never grow above 1 so stores cannot become visible out of order and ii) the store cannot be observed by the process itself before it is flushed. In effect this makes the store and flush operations atomic, reducing the program's behaviors to those possible under sequential consistency.

5.2 Fence Inference under Concrete Semantics

Buffers of Labeled Stores The semantics given in Section 3 do not preserve enough information about program execution to enable fence inference. Using those semantics, it is not possible to determine that a given memory operation was delayed by examining only the source state and the transition associated with the operation. Therefore, we instrument the concrete semantics with additional information about the instruction that stored each value. To achieve this, for a process p and variable x, we extend the store buffer $B_p(x) \in Seq(Labs \times D)$ to be a sequence of pairs $\langle l, v\rangle$. For every value stored we also record the label of the program instruction that stored the value.

Avoiding Error States Let P be a program, and $\langle \sigma_0, \Sigma, T\rangle$ be the program's transition system. Every transition $t \in T$ that is not a flush transition is associated with a instruction in the code that caused the transition. We denote by l_t the label of this instruction. Our goal is to construct a program P' by inserting fences into P such that the state-space of P' does not contain any error states. To "remove" a state from the state-space, we must prohibit all program traces that contain it. The question then becomes "how can a program trace be prohibited?"

The intuition behind the recipe of [40] is that program traces can be classified as either *avoidable* and *unavoidable*. The classification is performed according to the syntactic device we have to eliminate traces. In our setting, the syntactic device used to prohibit traces is a memory fence. If a trace π contains a store transition $\sigma_i \xrightarrow{t_p} \sigma_{i+1}$ by process p which is not immediately followed by a flush of the stored value, we can prohibit π by placing a fence immediately after the store. This means a trace is unavoidable if and only if every store is immediately followed by a flush.We can refine the concept of an avoidable trace and talk about *avoidable transitions*. A transition t_p performed by process p is avoidable if it is a memory operation (store, load or CAS), and some store buffer associated with p is non-empty.

Formally, let t be the transition $\sigma \xrightarrow{t} \tau$. Let v be a value written by the instruction at label l_v such that $\langle l, v\rangle$ appears in some buffer $B_{\sigma,p}(x)$. Then t is avoidable, and can be avoided by placing fences on all program paths between l_v and l_t, forcing the value v to be flushed before l_t is reached. We formalize this by defining *ordering constraints*: we say an ordering constraint $[l_v \prec l_t]$ is *enforced* if a fence is placed on all program paths between l_v and l_t. We say the constraint is *violated* by a transition if l_t is executed while a value stored by l_v is in the buffer. A constraint is violated by a trace if it is violated by one of its transition. Note that if a constraint is enforced by a fence, it cannot be violated by any transition. This implies a trace π of P cannot appear in P' if at least one of the constraints it violates is enforced in P'. This fact gives us a complete characterization of how a state can be removed from the state-space: by enforcing at least one constraint that is violated by each trace leading to that state. We call this characterization the *avoid formula* of a state.

A direct implementation of the method described above to compute the avoid formulae would be very inefficient, as it requires enumerating all program traces. Below we give a brief description of a more efficient algorithm. A fuller description, albeit in a different setting, is given in [18].

As a first stage in the algorithm we construct the transition system $\langle \sigma_0, \Sigma, T\rangle$. We then label every state $\sigma \in \Sigma$ with a propositional formula that captures how σ can be made unreachable (avoided) through the use of ordering constraints. Intuitively, a state σ can be avoided by avoiding *all* incoming transitions to σ in the program's transition system. In turn, a single transition $\mu \to \sigma$ can be avoided by either avoiding its source state μ or by prohibiting the transition itself. We associate with each transition $t \in T$ a formula:

$$prevent(t) = \bigvee\{[l \prec l_t] \mid \exists x, v. \langle l, v\rangle \in Set(B_{src(t),proc(t)}(x))\}$$

Here, we use $proc(t)$ to denote the process that executes the transition t and $src(t)$ to denote the source state of the transition. This formula captures all possible ordering constraints that would prohibit execution of t. Formally, it is a disjunction because it is enough to enforce one of the constraints to make t unreachable.

To compute how a state $\sigma \in \Sigma$ can be avoided, we define a labeling function L and:

$$avoid(L, \sigma) = \bigwedge\{(L(\mu) \vee prevent(t)) \mid t = (\mu \to \sigma) \in T\}$$

We then define a transformer that updates the labeling function:

$$infer(L) = L[\sigma \mapsto (L(\sigma) \wedge avoid(L, \sigma))]$$

Given an initial mapping L_0 that maps all unavoidable states to $false$ and the rest to $true$, the greatest fixed point of $infer(L_0)$ describes all the possible ways in which any state σ can be avoided. The greatest fixed point is computed with respect to implication partial order $L_1 \sqsubseteq L_2 \equiv \forall \sigma \in \Sigma. L_1(\sigma) \Rightarrow L_2(\sigma)$. Using the provided specification, we identify a set $E \subseteq \Sigma$ of reachable error states. We then compute the overall constraint formula ψ by taking the conjunction of avoid constraints for all error states: $\psi = \bigwedge\{L(\sigma) \mid \sigma \in E\}$. A satisfying assignment to this formula is guaranteed to represent a correct fence placement.

5.3 Inference under Abstract Semantics

We can extend the abstract model in the same way we extended the concrete model. That is, $H_p(x)$ and $S_p(x)$ will contain $\langle label, value\rangle$ pairs.

5.3.1 Inference under Disjunctive Abstraction

Using the abstract semantics of Sec. 4.2, we can construct an abstract transition system for the program, and apply the same reasoning as in the concrete semantics, except that we adjust $prevent(t)$:

$$Q_{\sigma,p}(x) = S_{\sigma,p}(x) \cup Set(H_{\sigma,p}(x))$$

$$prevent(t) = \bigvee\{[l \prec l_t] \mid \exists x, v.\langle l, v\rangle \in Q_{src(t),proc(t)}(x)\}$$

This adjustment is safe because we know that if $\langle l_v, v\rangle \in Q_p(x)$, then for any concretization $\sigma^\natural$ of σ, $B_{\sigma^\natural,p}(x)$ must contain $\langle l_v, v\rangle$ at least once. This means that placing a fence between any such l_v and l_t is sufficient to avoid t from $\sigma^\natural$.

Note that it is possible to infer more fences than necessary due to the imprecision of the abstraction. Consider the simple example in Fig. 6, with the specification that in a final state $r1 \leq r2$. If we attempt to execute this program under partial-coherence semantics with $k = 0$, we may get a trace where in the final state we have $r1 = 2$, $r2 = 1$: *(a)* Process 1 performs both stores. *(b)* Process 1 flushes the value 2. *(c)* Process 2 performs the load at line 1. *(d)* Process 1 flushes the value 1. *(e)* Process 2 performs the load at line 2. The single avoidable transition in this trace is the execution of the second store by process 1. The only way to avoid this transition is by placing a fence between the two stores. However, if we increase the precision of the abstraction and use $k = 1$, we will not produce this (spurious) trace and will not infer the redundant fence.

```
   Process 1:                       Process 2:

1  store x = 1;                  1  load r1 = x;
2  store x = 2;                  2  load r2 = x;
```

Figure 6. Fully disjunctive partial-coherence abstraction with k = 0 leads to a redundant fence between the stores in Process 1, while with k = 1 the inference algorithm determines that no fences are necessary.

5.3.2 Inference under Partially Disjunctive Abstraction

For the abstract semantics of Sec. 4.3, we need to adjust $prevent(t)$:

$$prevent(t) = \bigwedge\{[l \prec l_t] \mid \exists x, v.\langle l, v\rangle \in Q_{src(t),proc(t)}(x)\}$$

The only change from the fully disjunctive abstraction is in replacing $\bigvee$ with $\bigwedge$. The reason for this change becomes clear once we examine the concretization function for the partially disjunctive abstraction. As pointed out in the previous section, given an abstract state σ and a non-empty $S_{\sigma,p}(x)$, there exist concretized states which do *not* contain all values in $S_{\sigma,p}(x)$. Since prohibiting a transition from σ requires prohibiting that transition from all concrete states represented by σ, $prevent(t)$ must be a *conjunction* over the possible prevent formulas in the concrete domain. For many transitions, this formula will be stronger than the optimal one, potentially leading to a fence placement worse than the one produced by the fully disjunctive abstraction with the same k value.

5.4 Fine-grained fence inference

The inference algorithm described above generates sets of constraints that must be enforced so that the specification is satisfied. One simple way to enforce a constraint $[l_1 \prec l_2]$ is by placing a full fence on every path between l_1 and l_2 on the control-flow graph of the program. If finer-grained fences are available on the architecture we can use information encoded in the constraint to implement it more efficiently. For example if the architecture provides separate *store-store* and *store-load* fences we can place the appropriate fence based on whether the instruction at l_2 is a store or a load. If the architecture provides fences that enforce flushing only one variable (e.g., CAS in our concrete PSO semantics) then we can place the correct fence type based on the variable written to by l_1. For simplicity, in Section 6 we assume the only fence available is a full fence. However, whenever inference succeeds we could trivially place finer-grained fences.

6. Experience

We implemented our abstractions together with the verification and inference algorithms in a tool called BLENDER. Using BLENDER, we demonstrate the effectiveness of our abstractions by successfully verifying and inferring the required fences in a number of challenging algorithms. None of these algorithms could be handled by existing approaches. Further, we illustrate an inherent trade-off between the optimality of fence inference and the state-space size dictated by the abstraction.

BLENDER is implemented in Java and uses the JavaBDD library to represent avoid formulae as BDDs. All experiments were conducted on an 8-CPU Xeon 1.6GHz with 16GB memory running a 64-bit Sun JVM on Red Hat Linux.

Abstractions In our experiments, we consider a range of abstract memory models, all of which are abstractions of the concrete PSO memory model:

- *Set*: an abstraction of the store buffer to a set, without any additional information such as recency.
- *FD*: the partial coherence abstraction shown in Sem. 3, with varying k.
- *PD*: the partially disjunctive abstraction described in Sec. 4.3.

Note that the *Set* abstraction and *FD/PD* with $k = 0$ are generally not SC-Recoverable. Thus, it is possible that during fence inference, BLENDER will report the program as impossible to fix.

6.1 Benchmarks

To evaluate our tool, we chose various classic concurrent algorithms such as well-known mutual exclusion algorithms (mutex) and synchronization barrier algorithms. All algorithms were exercised in a loop by two concurrent processes ("repeated entry"):

- *Dekker's Algorithm* [11]. To evaluate both inference and verification we used two versions:
  - *Dek0*: has no added fences and is incorrect under the PSO memory model.
  - *Dek2*: has two added fences and is known to be correct.
- *Peterson's Algorithm* [31], using two versions, *Pet0* and *Pet2*.
- A variation of *Lamport's Bakery* [19] using two versions, *Lam0* and *Lam2*. To make this algorithm finite-space we manually bounded the maximum ticket number at 2.
- *Lamport's Fast Mutex* [21] using two versions, *Fast0* and *Fast3*.
- *CLH queue lock* [26] using two versions, *CLH0* and *CLH2*
- Centralized sense-reversing synchronization barrier [14] using two versions *Sense0* and *Sense1*.

For the mutual exclusion, the specification is that there cannot be more than one process inside the critical section. "Release semantics" for operations within the critical section are not enforced.
The benchmarks were selected based on two criteria:

Novelty The benchmarks could not be handled by any of the previous approaches. For instance, as mutual exclusion algorithms inherently contain benign data races, using techniques like delay set analysis [34] would result in a gross over-estimation of the required fences. Furthermore, some of the benchmarks — for instance *Dek* and *Fast* — contain benign *triangular* data races (as defined in [28]). Thus, even if we focus squarely on the TSO memory model, we could not use the results of [28] to establish the correctness of the algorithms by focusing only on sequentially consistent executions. Finally, all of the benchmarks contain unbounded spin-loops, and as such, they cannot be handled directly using the bounded techniques of [18] or [5].

Simplicity Our focus in this work has been abstracting the effect of the relaxed memory model in isolation from other sources of unboundedness. Hence, we chose our algorithms to be finite-state when executed under the SC model. We defer the problem of verifying infinite-state programs using abstract memory models (e.g., by composing our abstractions with heap or predicate abstractions) to future work.

6.2 Verification

Tab. 1 shows the verification results produced by BLENDER with three abstractions. The programs we used here are the ones we know to be correct under the *concrete* PSO semantics, that is, appropriate fences have been placed in advance. All verification runs completed within 30 seconds. Each entry in the table contains the total number of states explored (in thousands). A ● mark is placed if verification succeeded and a ○ mark if a spurious (abstract) counter-example was found. In some of the runs of *CLH2*, BLENDER exhausted the available memory, and thus we do not report the state-space size. However, in both those cases an (abstract) counter-example was found before BLENDER ran out of memory.

Prog.	Set	$FD_{k=0}$	$FD_{k=1}$
Sense1	● 0.6	● 1.7	● 0.8
Pet2	○ 7.7	● 2.4	● 1.8
Dek2	○ 9.7	● 4.5	● 3.1
Lam2	○ 41.2	● 22.2	● 9
Fast3	○ 22.2	○ 16.4	● 11.1
CLH2	○ M	○ M	● 68.8

Table 1. Verification results and number of states (in thousands).

6.2.1 Discussion

As Tab. 1 shows, none of the correct examples could be verified using the naive set abstraction, however all of them could be verified using *FD* with $k = 1$. Since verification of all examples successfully completed with *FD*, there was no need to use the *PD* abstraction. The table also shows $FD_{k=0}$ generated spurious counter-examples for *CLH2* and *Fast3* but not the other algorithms. When $k = 0$, the partial-coherence abstraction ($FD_{k=0}$) reduces to the set abstraction with recency information. This is enough to verify the simpler algorithms, however it fails on the more complex ones.

The example of Lamport's fast mutex is particularly interesting, as it demonstrates the type of executions possible with non SC-recoverable abstractions. Consider the code in Fig. 7. In this implementation a process can enter the critical section either along the fast path (the `if` condition in line 15 is false) or along the slow path (the conditions is true). Under an abstract model with $k = 0$, the following execution is possible:

- Process 1 enters the critical section along the fast path, executes it, and runs until line 29.
- Process 1 executes line 29. At this point $S_1(y) = \{0\}$.
- Process 1 flushes y non-destructively, using the `FLUSH-SN` rule. Now $G(y) = 0$.
- Process 2 enters the critical section. Since $G(y) = 0$ it enters along the fast-path setting $y = 2$ in the process. This is flushed destructively using the `FLUSH-SD` rule. At this point $G(y) = 2$, $S_1(y) = \{0\}$, $S_2(y) = \emptyset$.
- Process 1 resumes. It first performs a flush of y, setting $G(y) = 0$. Then it proceeds to enter the critical section again, using the fast path.

This execution relies on the fact p_1 only stored the value 0 to y once, but this store is flushed twice. In effect, p_2 observed this store as if it happened *before* its own, and p_1 observed it as if it happened *after* the store of p_2. This coherence violation would have been prevented if we kept more information in the content of the buffer, by using $k > 0$. Indeed, with $k = 1$, *Fast3* passes verification.

```
1  while(true)
2  {
3    start:
4    store b[i] = true;
5    store x = i;
6    load local_y = y;
7    if (local_y ≠ 0) {
8      store b[i] = false;
9      while(local_y ≠ 0)
10       load local_y = y;
11     goto start;
12   }
13   store y = i;
14   load local_x = x;
15   if (x ≠ i) {
16    store b[1] = false;
17    do {
18     load other_b = b[3-i];
19    } while (other_b ≠ 0);
20    load local_y = y;
21    if (local_y ≠ i)
22    {
23      while(local_y ≠ 0)
24        load local_y = y;
25      goto start;
26    }
27   }
28   //Critical Section
29   store y = 0;
30   store b[i] = false;
31 }
```

Figure 7. A version of Lamport's fast mutex algorithm for 2 processors. The code given is for process i.

6.3 Inference

In Tab. 2 we show the state-space size and inference results for 5 of the under-fenced implementations. A mark of ● means the optimal fences were inferred, ◐ means that sub-optimal fences were inferred, and ○ means that BLENDER was unable to infer fences as according to the analysis any fence placement would leave the program incorrect. M appears if BLENDER ran out of memory.

Prog.	$FD_{k=0}$	$FD_{k=1}$	$PD_{k=0}$	$PD_{k=1}$	$PD_{k=2}$
Sense0	◐ 57.6	● 58.6	◐ 31.9	● 44.3	● 6.1
Pet0	● 69.2	● 524.5	◐ 25.3	◐ 244.7	● 1124.2
Dek0	● 424.2	● 3238.1	◐ 16.5	◐ 358.1	● 2350.0
Lam0	M	M -	◐ 421.0	◐ 4045.9	M -
Fast0	M -	M -	○ 220.6	M -	M -
Fast1a	◐ 139.0	● 105.2	○ 33.0	● 88.1	● 41.9
Fast1b	◐ 832.9	● 972.3	○ 78.6	◐ 501.6	● 878.4
Fast1c	M -	M -	○ 110.4	◐ 1173.4	● 1858.1
CLH0	M -	M -	M -	M -	M -

Table 2. Inference results and number of states (in thousands).

6.3.1 Discussion

Initially, we used BLENDER to perform fence inference with abstractions $FD_{k=0}$ and $FD_{k=1}$. However, BLENDER ran out of memory for *Lam0*, *Fast0*, and *Fast1c*. Using the partially disjunctive abstraction $PD_{k=0}$ enabled us to run the inference algorithm for both *Lam0* and *Fast1c* and obtain a sound fence placement for both. Furthermore, despite the loss of precision in the PD abstraction, in both cases the inferred fences are not trivial.

6.3.2 Peterson's Algorithm

Our results for Peterson's algorithm demonstrate the inherent trade-offs between inference optimality and abstraction precision:

- With the FD abstraction BLENDER was able to infer the optimal fence placement with $k = 0$. With the PD abstraction it required $k = 2$ and a much larger state-space.
- With the $PD_{k=0}$ abstraction we can produce a smaller state space but the result is suboptimal: 3 fences are required instead of 2. In addition to the two fences shown in Fig. 2, another fence, immediately after the store in line 14, is inferred.

The same trade-off can also be observed when using a similar partial-coherence abstraction of the *TSO* model. For $k = 0$ and $k = 1$ suboptimal fence placement is generated, while with $k = 2$ the result is optimal (for TSO).

6.3.3 Lamport's Fast Mutex

For both *Fast0* and *Sense0*, we experienced a loss of precision when using a k value that is too small. In the case of *Fast0*, the inference algorithm reported the program as unfixable when using $PD_{k=0}$. This is due to the fact the counter-example presented for *Fast3* under this abstract model cannot be fixed with any number of fences. Unfortunately, BLENDER was unable to build the state-space of *Fast0* under $PD_{k=1}$. Thus, we've run a complementary set of experiments in which 1 of the 3 required fences was placed. The 3 versions of Lamport's fast mutex (Fig. 7) we have ran had a single fence inserted: (i) between lines 5 and 6 (*Fast1a*), (ii) between lines 13 and 14 (*Fast1b*), (iii) between lines 29 and 30 (*Fast1c*). As expected, for all 3 programs, when running under $PD_{k=0}$ the program was unfixable. However, in all 3 cases we were able to infer a correct fence placement using $PD_{k=1}$. Furthermore, for *Fast1a* and *Fast1b* the optimal placement of the two other fences was found when using $PD_{k=2}$. For *Fast1c* even with $k = 2$ the placement was still suboptimal. This demonstrates another example of the interplay between the placed fences and the precision of the required abstraction. Even though for *Fast1c* we could not infer the optimal fence placement using $PD_{k=1}$, had we placed them manually, this abstraction could be used to verify them.

7. Related Work

Data-Race Freedom Guarantee A common technique to reduce the complexity of analyzing programs under relaxed memory models is to focus only on programs that have no data-races under sequentially consistent executions. For such programs the "fundamental property of memory models" [32] (also known as the DRF theorem) ensures that there can be no sequentially inconsistent executions. Owens (in [28]) studies a generalization of this theorem for the x86-TSO model. To guarantee correctness under this model one needs only to prove sequentially consistent executions satisfy "triangular race-freedom", a property weaker than general data race-freedom. In our work, we focus on abstractions of arbitrary programs and unlike these methods, we can handle programs that contain data-races, such as common lock-free algorithms and mutual exclusion primitives.

Checking Equivalence To Sequential Consistency In [6] and [7] algorithms are presented that can, based only on sequentially consistent executions, find violations of sequential consistency under the TSO and PSO memory models. Similarly, it is possible to place fences to preserve only SC executions using *Delay Set Analysis* ([34]), for instance as implemented in the Pensieve compiler ([12, 22]). However, a violation of SC does not necessarily cause a violation of any high-level properties. Thus those algorithms are often needlessly conservative. Our approach, on the other hand, uses a high-level specification and allows a trade-off between precision and optimality of the solution.

Explicit Model Checking for Relaxed Memory Models [17, 18, 30] describe explicit-state model checking under the Sparc RMO model. Among those, [18] focuses on fence inference. [15] also describes an explicit-state model checking and inference technique for the .NET memory model, but it suffers from significant technical drawbacks (c.f. [18]). However, the techniques presented in [18] are not applicable in our setting. (i) The FENDER algorithm described in [18] can only infer fences for programs that are finite-state *under the relaxed memory model*, and not under the sequentially consistent model. While this distinction might seem subtle, it is in fact significant. For example, FENDER relies on finite clients of lock-free data structures being finite-state. Unfortunately, there is no guarantee that a data structure that is lock-free under SC will stay lock-free under a relaxed model. More generally, any code that uses a spin-loop with a store in the loop body will always be infinite-state under a relaxed model, unless the store is followed by a fence. Since classical implementations of synchronization primitives use this code pattern, it is not possible to use FENDER to infer fences in those implementations. In contrast, the technique we present in this paper requires the input to be finite-state *only under SC*. (ii) More technically, the algorithm is phrased in terms of execution buffer semantics. Adapting it to store-buffer semantics is challenging, especially in the abstract case.

In [5], Burckhardt et. al take a different approach to verification under RMM. Instead of working with operational memory models and explicit model-checking, they convert programs into a form that can be checked against an axiomatic model specification. This technique still suffers from the same limitation — it must unroll loops at a preprocessing stage. Thus it cannot verify programs that contain unbounded spinning. In contrast, our verification approach is based on abstract interpretation and is sound for any input program and any values of the abstraction parameters. An alternative approach, in the spirit of [5, 18], is to assume a bound on the size of buffers or the number of loop iterations. Combined with iterative increase of the bound, this may work for some examples — but not in the general case. In addition, using an abstraction is beneficial not only when the buffers are unbounded. Even if the buffer is bounded, the concrete state-space may simply be too large — while in fact representing the buffers with full precision is not important to the correctness of the algorithm.

Synchronization Inference In [38, 40], the authors propose algorithms that automatically infer synchronization constructs such as atomic sections and conditional critical regions. These works assume sequential consistency and do not support weak memory models. The approach of [38] is close to our work in spirit, and deals with inferring synchronization under abstraction, but it enumerates traces explicitly, which does not scale to our setting.

In [37, 39], inference of synchronization is performed by syntactic exploration of placements of atomic sections to create candidate algorithms, and using a backing verifier to attempt verification of each candidate. In principle, a similar approach can be employed for fence inference by exploring candidate algorithms with all possible fence placements. In contrast, our constraint-based approach lays the ground for inference of more advanced fences, such as fence per variable, and conditional fences, for which syntactic exploration will yield a non-feasible number of candidates.

Alternative Buffer Abstractions In [23] the authors use automata as symbolic representation of store buffers in the TSO memory model. Their approach uses an acceleration technique that does not guarantee termination. Furthermore, their automata-based representation preserves redundant information, and as noted by the authors themselves, ends up being too expensive to be of practical interest. Since store buffers are similar to FIFO channels in communicating FSMs (CFSMs), it is tempting to employ techniques from CFSMs in our context as well. These techniques include algorithms based on symbolic representation of channel content (e.g., [4]), and conservative abstractions for FIFO channels (e.g., [13]). Abstraction of FIFO channels, as presented in [13], is similar in spirit to our approach in that it guarantees termination by using approximation. However, their abstraction preserves a slightly different kind of information than the information required for reasoning about store buffers. They use a regular abstraction of queue content and an expensive widening operation to establish correct usage of protocols. This is more than what is required in our setting in terms of characterization of buffer content, and often less than needed in terms of recency information. In contrast to these, our technique guarantees termination by using conservative approximation, and our abstractions are tailored to relaxed memory models.

8. Conclusions and Future Work

We present an approach for automatic verification and fence inference for concurrent programs running under relaxed memory models. Our approach is based on abstract interpretation, and its technical core is a family of partial-coherence abstractions that provide a (parametric) bounded representation for potentially unbounded store buffers. Our abstractions enable us to automatically verify concurrent algorithms without worrying about the size of the underlying store buffers. Because partial coherence abstractions are designed to be SC-Recoverable, they can be used for automatic inference of memory fences. We have implemented our approach in a tool called BLENDER and applied it to verify several correctly-fenced concurrent algorithms and automatically infer fences in under-fenced versions of these algorithms. In the future, we plan to combine our abstractions with heap abstractions to enable verification of heap-manipulating programs under RMMs.

Acknowledgments

The authors wish to thank Noam Rinetzky and Greta Yorsh for their comments on earlier drafts of this paper. This research was partially supported by *The Israeli Science Foundation* (grant no. 965/10).

References

[1] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *IEEE Computer 29* (1995), 66–76.

[2] ATIG, M. F., BOUAJJANI, A., BURCKHARDT, S., AND MUSUVATHI, M. On the verification problem for weak memory models. In *POPL* (2010), pp. 7–18.

[3] BOEHM, H.-J. Threads cannot be implemented as a library. *SIGPLAN Not. 40*, 6 (2005), 261–268.

[4] BOIGELOT, B., GODEFROID, P., WILLEMS, B., AND WOLPER, P. The power of QDDs. In *SAS* (1997), Springer, pp. 172–186.

[5] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI* (2007), pp. 12–21.

[6] BURCKHARDT, S., AND MUSUVATHI, M. Effective program verification for relaxed memory models. In *CAV* (2008), pp. 107–120.

[7] BURNIM, J., SEN, K., AND STERGIOU, C. Sound and complete monitoring of sequential consistency in relaxed memory models. Tech. Rep. UCB/EECS-2010-31.

[8] BURNIM, J., SEN, K., AND STERGIOU, C. Testing concurrent programs on relaxed memory models. Tech. Rep. UCB/EECS-2010-32.

[9] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL* (1977), pp. 238–252.

[10] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *POPL* (1979), pp. 269–282.

[11] DIJKSTRA, E. Cooperating sequential processes, TR EWD-123. Tech. rep., Technological University, Eindhoven, 1965.

[12] FANG, X., LEE, J., AND MIDKIFF, S. P. Automatic fence insertion for shared memory multiprocessing. In *ICS* (2003), pp. 285–294.

[13] GALL, T. L., JEANNET, B., AND JRON, T. Verification of communication protocols using abstract interpretation of FIFO queues. In *AMAST* (2006), pp. 204–219.

[14] HENSGEN, D., FINKEL, R., AND MANBER, U. Two algorithms for barrier synchronization. *Int. J. Parallel Program. 17*, 1 (1988), 1–17.

[15] HUYNH, T. Q., AND ROYCHOUDHURY, A. Memory model sensitive bytecode verification. *Form. Methods Syst. Des. 31*, 3 (2007).

[16] IBM. *Power ISA v.2.05.* 2007.

[17] JONSSON, B. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News 36*, 5 (2008), 65–71.

[18] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. In *FMCAD* (2010), pp. 111–119.

[19] LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (1974), 453–455.

[20] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[21] LAMPORT, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst. 5*, 1 (1987), 1–11.

[22] LEE, J., AND PADUA, D. A. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput. 50*, 8 (2001), 824–833.

[23] LINDEN, A., AND WOLPER, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN* (2010), pp. 212–226.

[24] MADOR-HAIM, S., ALUR, R., AND MARTIN, M. M. K. Generating litmus tests for contrasting memory consistency models. In *CAV* (2010), pp. 273–287.

[25] MADOR-HAIM, S., ALUR, R., AND MILO, M. Plug and Play Components for the Exploration of Memory Consistency Models. Tech. Rep. MS-CIS-10-02, University of Pennsylvania, 2010.

[26] MAGNUSSON, P. S., LANDIN, A., AND HAGERSTEN, E. Queue locks on cache coherent multiprocessors. In *Proceedings of the Int. Symp. on Parallel Processing* (1994), IEEE, pp. 165–171.

[27] NARAYANASAMY, S., WANG, Z., TIGANI, J., EDWARDS, A., AND CALDER, B. Automatically classifying benign and harmful data races using replay analysis. In *PLDI* (2007), pp. 22–31.

[28] OWENS, S. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP* (2010).

[29] OWENS, S., SARKAR, S., AND SEWELL, P. A better x86 memory model: x86-TSO. In *TPHOLs* (2009), pp. 391–407.

[30] PARK, S., AND DILL, D. L. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers 48* (1999).

[31] PETERSON, G. L. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (1981), 115–116.

[32] SARASWAT, V. A., JAGADEESAN, R., MICHAEL, M., AND VON PRAUN, C. A theory of memory models. In *PPoPP* (2007), ACM, pp. 161–172.

[33] SARKAR, S., SEWELL, P., NARDELLI, F. Z., OWENS, S., RIDGE, T., BRAIBANT, T., MYREEN, M. O., AND ALGLAVE, J. The semantics of x86-cc multiprocessor machine code. In *POPL* (2009), pp. 379–391.

[34] SHASHA, D., AND SNIR, M. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst. 10*, 2 (1988), 282–312.

[35] SHEN, X., ARVIND, AND RUDOLPH, L. Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. *SIGARCH Comput. Archit. News 27*, 2 (1999), 150–161.

[36] SPARC INTERNATIONAL, INC. *The SPARC architecture manual (version 9).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[37] VECHEV, M., AND YAHAV, E. Deriving linearizable fine-grained concurrent objects. In *PLDI* (2008), pp. 125–135.

[38] VECHEV, M., YAHAV, E., AND YORSH, G. Abstraction-guided synthesis of synchronization. In *POPL* (2010), pp. 327–338.

[39] VECHEV, M. T., YAHAV, E., BACON, D. F., AND RINETZKY, N. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI* (2007), pp. 456–467.

[40] VECHEV, M. T., YAHAV, E., AND YORSH, G. Inferring synchronization under limited observability. In *TACAS* (2009), pp. 139–154.

[41] YANG, Y., GOPALAKRISHNAN, G., AND LINDSTROM, G. UMM: an operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper. 17*, 5-6 (2005), 465–487.