Thread-Modular Static Analysis for Relaxed Memory Models

Markus Kusano
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

We propose a memory-model-aware static program analysis method for accurately analyzing the behavior of concurrent software running on processors with weak consistency models such as x86-TSO, SPARC-PSO, and SPARC-RMO. At the center of our method is a unified framework for deciding the feasibility of inter-thread interferences to avoid propagating spurious data flows during static analysis and thus boost the performance of the static analyzer. We formulate the checking of interference feasibility as a set of Datalog rules which are both efficiently solvable and general enough to capture a range of hardware-level memory models. Compared to existing techniques, our method can significantly reduce the number of bogus alarms as well as unsound proofs. We implemented the method and evaluated it on a large set of multithreaded C programs. Our experiments show the method significantly outperforms state-of-the-art techniques in terms of accuracy with only moderate runtime overhead.

CCS CONCEPTS

• **Software and its engineering** → *Automated static analysis*; *Formal software verification*;

KEYWORDS

Concurrency, Abstract interpretation, Thread-modular reasoning, Datalog, Relaxed memory model, TSO, PSO, RMO

ACM Reference format:
Markus Kusano and Chao Wang. 2017. Thread-Modular Static Analysis for Relaxed Memory Models. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106243

1 INTRODUCTION

Concurrent software written for modern computer architectures, though ubiquitous, remains challenging for static program analysis. Although abstract interpretation [13] is a powerful static analysis technique and prior *thread-modular* methods [19, 40, 47–49] mitigated *interleaving explosion*, none was specifically designed for software running on weakly consistent memory. This is a serious deficiency since weakly consistent memory may exhibit behaviors not permitted by uniprocessors. For example, slow memory accesses may be delayed, increasing performance, but also introducing additional inter-thread non-determinism. Thus, multithreaded software running on such processors may exhibit erroneous behaviors not manifesting on sequentially consistent (SC) memory.

```
void thread1() {                void thread2() {
  x = 1;                          y = 1;
  a = y;                          b = x;
}                               }
            assert( !(a == 0 && b == 0) );
```

Figure 1: The assertion holds under SC, but not under x86-TSO, SPARC-PSO, and SPARC-RMO memory models.

Consider x86-TSO (total store order) as an example. Under TSO, each processor has a *store buffer* caching memory write operations so they do not block the execution of subsequent instructions [4]. Conceptually, each processor has a queue of pending writes to be flushed to memory at a later time. The flush occurs non-deterministically at any time during the program's execution. This delay between the time a write instruction executes and the time it takes effect may cause the write to appear reordered with subsequent instructions within the same thread. Figure 1 shows an example where the assertion holds under SC but not TSO. Since x and y are initialized to 0 and they are *not* defined as *atomic* variables, the write operations (x=1 and y=1) may be stored in buffers, one for each thread, and thus delayed after the read operations.

SPARC-PSO (partial store order) permits even more non-SC behaviors: it uses a separate store buffer for each memory address. That is, x=1 and y=1 within the same thread may be cached in different store buffers and flushed to memory independently. This permits the reordering of a write to x with a subsequent read from y, but also with a subsequent write (e.g., to variable z) in the same thread. The situation is similar under SPARC-RMO (relaxed-memory order). We detail how such relaxation leads to errors in Section 2.

Broadly speaking, existing *thread-modular* abstract interpreters fall into two categories, neither modeling weak-memory related behaviors. The first are SC-specific [17, 18, 40]: they are designed to be flow-sensitive in terms of modeling thread interactions but consider only behaviors compatible with the SC memory. The second [47–49] are oblivious to memory models (MM-oblivious): they permit all orderings of memory-writes across threads. Therefore, MM-oblivious methods may report spurious errors (bogus alarms) whereas SC-specific methods, although more accurate for SC memory, may miss real errors on weaker memory (bogus proofs). This flaw is not easy to fix using conventional approaches [49]. For example, maintaining relational invariants at all program points makes the analysis prohibitively expensive. In Section 2, we use examples to illustrate issues related to these techniques.

We propose the first thread-modular abstract interpreter for analyzing concurrent programs under weakly consistent memory. Our method models thread interactions with flow-sensitivity, and is memory-model specific: it models memory operations assuming a processor-level memory model, as shown in Figure 2. In this figure, the boxes with bold text highlight our main contributions.

ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106243

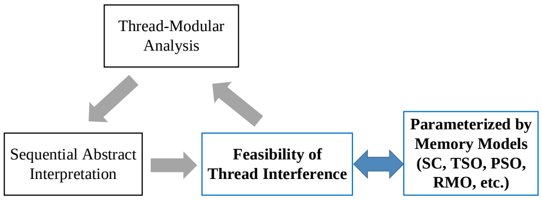
Thread-Modular Analysis
Sequential Abstract Interpretation
Feasibility of Thread Interference
Parameterized by Memory Models (SC, TSO, PSO, RMO, etc.)

Figure 2: FruitTree: Our memory-model-aware, thread-modular, static program analysis procedure.

Our method builds on a unified framework for modeling the memory consistency semantics. Specifically, the feasibility of thread interactions is formulated as a constraint problem via Datalog: it is efficiently solvable in polynomial time, and adaptable to various hardware-level memory models. Additionally, our method handles thread interactions in a flow-sensitive fashion while being thread-modular. Analyzing one thread at a time, as opposed to the entire program, increases efficiency, especially for large programs. However, unlike prior MM-oblivious methods we do not join all the effects of remote stores before propagating them to a thread, thus preserving accuracy. Overall, our method differs from the state-of-the-art, which either are *non-thread-modular* [14, 17, 39, 45] or not specifically targeting weak memory [40, 47–49].

Our method also differs significantly from techniques designed for bug hunting as opposed to obtaining correctness proofs. For example, in concurrency testing, stateless dynamic model checking [22, 25] was extended from SC to weaker memory models [1, 2, 16, 32, 51, 52, 67]. In bounded model checking, Alglave et al. [8] modeled weak memory through code transformation or direct symbolic encoding [7, 9]. However, these methods cannot be used to verify properties: if they do not find bugs, it does not mean the program is correct. In contrast, our method, like other abstract interpreters, is geared toward obtaining correctness proofs.

We implemented our new method in a tool named FruitTree, using Clang/LLVM [5] as the C front-end, Apron [37] for abstract domains, and the μZ [30] Datalog engine in Z3 [15]. We evaluated FruitTree on 209 litmus tests, and 52 larger multithreaded programs totaling of 61,981 lines of C code. Reachability properties were expressed as embedded assertions. Our results show that FruitTree is significantly more accurate than state-of-the-art techniques with moderate runtime overhead.

Specifically, we compared FruitTree against the MM-oblivious analyzer of Miné [49], the SC-specific thread-modular analyzer Watts [40], and a non thread-modular analyzer named Duet [17, 18]. On the litmus tests, FruitTree is more accurate than the other three methods. On the larger benchmarks, including Linux device drivers, FruitTree proved 4,577 properties, compared to 1,752 proved by Miné's method.

To summarize, we make the following contributions:

- We propose a memory-model aware static analysis method based on thread-modular abstract interpretation.
- We introduce a declarative analysis framework for deducing the feasibility of thread-interferences on weak memory.
- We implement and evaluate our method on a set of benchmarks to demonstrate its high accuracy and moderate runtime overhead.

The remainder of this paper is organized as follows. First, we motivate our technique via examples in Section 2. Then, we provide background on memory models and abstract interpretation in Section 3. We present our new declarative analysis for checking the feasibility of thread inferences in Section 4, followed by the main algorithm for thread-modular abstract interpretation in Section 5. We present our experimental results in Section 6, review related work in Section 7, and conclude in Section 8.

```
void thread1() {
  x = 5;
  fence;
  y = 10;
}
```

```
void thread2() {
  if (y == 10) {
    assert(x == 5);
  }
}
```

Figure 3: The assertion always holds, but if the fence is removed, the assertion may fail under PSO and RMO.

2 MOTIVATION

Consider the program in Figure 3. The assertion holds under SC, TSO, PSO, and RMO. But, removing the fence causes it to fail under PSO and RMO. In this section, we show why MM-oblivious methods may generate bogus errors, why SC-specific ones may generate bogus proofs, and how our new method fixes both issues.

2.1 Behaviors under SC, TSO, PSO, and RMO

First, note that the assertion in Figure 3 holds under SC since each thread executes its instructions in program order, i.e., `x = 5` *takes effect* before `y = 10`. So, thread two observing `y` to be 10 implies `x` must have been set to 5.

Next, we explain why the assertion holds under TSO [4]. TSO permits the delay of a store after a subsequent load to a disjoint memory address (as in Figure 1). This *program-order relaxation* is a performance optimization, e.g., buffering slow stores to speed up subsequent loads. However, since all stores in a thread go into the same buffer, TSO does not allow the reordering of two stores (thread 1, Figure 3). Thus, even without the fence, `x = 5` always *takes effect* before `y = 10`, meaning the assertion holds.

Next, we show why removing the fence causes the assertion to fail under PSO and RMO. Both permit store–store reordering by allowing each processor to have a separate store buffer for each memory address. Thus `x` and `y` are in separate buffers. Since buffers are flushed to memory independently, with the fence removed, `y = 10` may take effect before `x = 5`, as if the two instructions were reordered in this thread. Thus, the second thread may read 10 from `y` before 5 is written to `x` in global memory, thus causing the assertion to fail.

The fence is important because it forces all stores issued before the fence to be visible to all loads issued after the fence, i.e., `x = 5` takes effect before `y = 10`, even under PSO and RMO. Thus, the assertion holds again.

2.2 Ineffectiveness of Existing Methods

MM-oblivious methods [47–49] report bogus alarms because they were not designed for weak memory, and they ignore the causality of inter-thread data flows. Thus, they tend to drastically over-approximate the interferences between threads.

For example, an MM-oblivious static analysis may work as follows. First, it analyzes each thread as if it were a sequential program. Then, it joins the effects of all stores on global memory—known as the *thread interferences*. Next, it individually analyzes each thread again, this time in the presence of the thread interferences computed from the previous iteration: when a thread performs a memory read,

Method	Program in Figure 1						Program in Figure 3					
	with fences			without fences			with fences			without fences		
	SC	TSO	PSO/RMO	SC	TSO	PSO/RMO	SC	TSO	PSO/RMO	SC	TSO	PSO/RMO
MM-oblivious (e.g. [47–49])	(bogus) alarm	(bogus) alarm	(bogus) alarm	(bogus) alarm	**alarm**	**alarm**	(bogus) alarm	(bogus) alarm	(bogus) alarm	(bogus) alarm	(bogus) alarm	**alarm**
SC-specific (e.g. [17, 18, 40])	**proof**	(bogus) proof	(bogus) proof	**proof**	(bogus) proof	(bogus) proof	**proof**	(bogus) proof	(bogus) proof	**proof**	(bogus) proof	(bogus) proof
Our method	**proof**	**proof**	**proof**	**proof**	**alarm**	**alarm**	**proof**	**proof**	**proof**	**proof**	**proof**	**alarm**

Figure 4: Comparing the effectiveness of various methods in handling the example programs in Figure 1 and Figure 3.

the value may come from any one of these thread interferences. This iterative process repeats until a fixed point is reached.

Next, we demonstrate how the MM-oblivious analyzer works on Figure 3. Consider the thread interferences to be a map from variables to abstract values in the interval domain [13]. Thread 1 generates interferences x $\mapsto$ $[5, 5]$ and y $\mapsto$ $[10, 10]$. Within thread 2, the load of y may read from local memory, $[0, 0]$, or the interference $[10, 10]$. Thus, y $= [0, 0] \sqcup [10, 10] = [0, 10]$, where $\sqcup$ is the *join* operator in the interval domain. Similarly, the load of x may read from local memory, $[0, 0]$, or the interference $[5, 5]$, i.e., x $= [0, 5]$. Thus, the assertion is incorrectly reported as violated.

While our previous example used the *non-relational* interval domain the bogus alarm remains when using a relational abstract domain: the propagation of interferences in MM-oblivious methods is inherently non-relational. Inferences map variables to a single values, causing all relations to be forgotten. Conventional approaches cannot easily fix this since maintaining relational invariants at all global program points is prohibitively expensive.

In contrast, prior SC-specific methods [17, 18, 40] do not report bogus alarms: they assume `x = 5` takes effect before `y = 10`. This leads to more accurate analysis results for SC, but is unsound under weak memory, e.g., they miss the assertion failure in Figure 3 under PSO or RMO when the fence is removed.

Figure 4 summarizes the ineffectiveness of prior techniques on the programs in Figures 1 and 3 with and without fences. Note that in Figure 1 the fence instruction may be added between the write and read instructions of both threads. The table in Figure 4 shows how prior MM-oblivious methods report bogus alarms, prior SC-specific methods report bogus proofs, while our new method eliminates both.

2.3 How Our Method Handles Memory Models

Some prior techniques lead to bogus alarms because they over-approximate thread interferences, i.e., they allow a load to read from any remote store regardless of whether such a data flow, or combination of flows, is feasible, while others lead to missed bugs because they under-approximate thread interferences, i.e., they do not allow any non-SC data flow. Consider Figure 3: the load of x may read 0 or 5, and the load of y may read 0 or 10, but the combination of x reading 0 and y reading 10 is infeasible. Realizing this, our method checks the feasibility of interference combinations under weak-memory semantics before propagating them.

Toward this end, we propose two new techniques. The first is the flow-sensitive propagation of thread interferences. Instead of eagerly joining all interfering stores, we handle each combination separately. The second is a declarative modeling of the memory consistency semantics general enough to capture SC, TSO, PSO, and RMO [4, 58, 65]. Together, these techniques prune infeasible combinations of thread interferences such as x and y reading 0 and 10, respectively, in Figure 3.

Our new method analyzes thread 2 in Figure 3 by considering four different interference combinations, ρ_1–ρ_4, separately.

- ρ_1 = y $\mapsto [0, 0]$ and x $\mapsto [0, 0]$,
- ρ_2 = y $\mapsto [10, 10]$ and x $\mapsto [5, 5]$,
- ρ_3 = y $\mapsto [0, 0]$ and x $\mapsto [5, 5]$,
- ρ_4 = y $\mapsto [10, 10]$ and x $\mapsto [0, 0]$.

We gain accuracy in two ways. First, we remove spurious values caused by an eager join (e.g., we no longer have y $= [0, 10]$). Second, we query a lightweight constraint system to quickly deduce infeasibility of an interference combination on demand. ρ_1, ρ_2, and ρ_3 are all feasible but they do not cause assertion failures.

Our check for infeasibility of an interference combination is implemented using Datalog (Horn clauses within finite domains), solvable in polynomial time. We will provides details of this constraint system in Section 4. For now, consider ρ_4 in Figure 3: it is infeasible (unless we assume the program runs under PSO or RMO with the fence removed). We deduce infeasibility as follows:

- `y = 10` has executed (it is being read from),
- thus `x = 5` has executed (due to the *program-order* requirement on SC and TSO, and the fence on PSO and RMO),
- so the load of x must not read from its initial value $[0, 0]$.

This deduction leads to a formal proof that ρ_4 can not exist in any concrete execution. Since the combinations ρ_{1-3} do not violate the assertion, and ρ_4 is proved to be infeasible, the property is verified.

3 PRELIMINARIES

In this section, we review weak memory models at the processor level (as opposed to the programming language level) and static program analysis based on abstract interpretation.

3.1 Concurrent Programs

We are concerned with a program consisting of a finite set of threads. Each thread assesses a set $\{a, b, c, \ldots\}$ of local variables. All threads access a set $\{x, y, z, \ldots\}$ of global variables via *load* and *store* instructions. A thread creates a child thread with *ThreadCreate*, and waits it to terminate with *ThreadJoin*.

We represent a program using a set $\mathbb{G} = \{G_1, \ldots, G_k\}$ of flow graphs. Each flow graph $G \in \mathbb{G}$, where $G = \langle N, n_0, \delta \rangle$, is a thread: $N \subseteq \mathbb{N}$ is the set of program locations of the thread, $n_0 \in N$ is the entry point, and δ is the transition relation. That is, $(n', n) \in \delta$ iff there exists an edge from n' to n.

Each program location $n \in \mathbb{N}$ is associated with an atomic instruction that may be a *load*, *store*, or *fence*. Non-atomic statements such as `y = x+1`, where both x and y are global variables, can be transformed to a sequence of atomic instructions, e.g., the load `a = x` followed by the store `y = a+1`, where a is a local variable in both

Memory Model	Which Program-Order Relaxation Is Allowed?								Write-Atomicity
	R(v_1)→R(v_1)	R(v_1)→W(v_1)	W(v_1)→R(v_1)	W(v_1)→W(v_1)	R(v_1)→R(v_2)	R(v_1)→W(v_2)	W(v_1)→R(v_2)	W(v_1)→W(v_2)	read own write early
SC [43]	no	no	no	no	no	no	no	no	no
TSO [55, 65]	no	no	no*	no	no	no	yes	no	yes
PSO [65]	no	no	no*	no	no	no	yes	yes	yes
RMO [58, 65]	no	no	no*	no	yes	yes	yes	yes	yes

Figure 5: Allowed relaxations of various processor-level memory models (cf. [4]). v_1 and v_2 are distinct variables, and * indicates rule needs to be relaxed to allow *read-own-write-early* behaviors (see Section 4.6 for explanation).

cases. When accessing variables on the global memory, threads may use a special *fence* instruction to impose a strict program order between memory operations issued before and after the fence.

3.2 Memory Consistency Models

The simplest memory model is sequential consistency (SC) [43]. SC corresponds to a system running on a single coherent memory time-shared by operations executed from different threads. There are two important characteristics of SC: the *program-order* requirement and the *write-atomicity* requirement. The program-order requirement says that the processor must ensure that instructions within a thread take effect in the order they appear in the program. The write-atomicity requirement says that the processor must maintain the illusion of a single sequential order among operations from all threads. That is, the effect of any store operation must take effect and become visible either to *all* threads or to *none* of the threads.

SC is an ideal memory model: In real CPUs, the hardware-level memory models are often weaker than SC, and can be characterized by their corresponding relaxations of the *program-order* and *write-atomicity* requirements as shown in Figure 5. Here, $R(v_1) \longrightarrow W(v_2)$ is a read of v_1 followed by a write of v_2 in the same thread.

Specifically, TSO allows x=1;a=y to be reordered as a=y;x=1, according to W(v_1)⟶R(v_2) in Column 8, where v_1 is x and v_2 is y. PSO further allows x=1;y=2 reordered to y=2;x=1, according to W(v_1)⟶W(v_2) in Column 9. As shown in Section 2, these program-order relaxations, conceptually, are the effect of store buffering, which delay the stores past subsequent stores/loads within a thread. Neither TSO nor PSO permits the delay of a load. Weaker still is RMO, which permits the relaxations of R(v_1)⟶R(v_2) and R(v_1)⟶W(v_2), as shown in Columns 6 and 7 of the table in Figure 5.

By relaxing the write-atomicity requirement, all three weaker memory models allow a thread to read its own write early. That is, the thread can read a value it has written before the value reaches the global memory and hence becomes visible to other threads.

3.3 Abstract Interpretation

Abstract interpretation is a popular technique for conducting static program analysis [13]. In this context, a numerical abstract domain defines, for every $n \in \mathbb{N}$ of the program, a memory environment s. It is a map from each program variable to its abstract value[1]. Consider intervals, which map each variable to a region defined by the lower and upper bounds. For a program with two integer variables x and y where both may have any value initially, the memory environment associated with the entry point $n_0 \in \mathbb{N}$ is $s_0 = \{x \mapsto \top, y \mapsto \top\}$, where $\top = (-\infty, +\infty)$. After executing x = 1, the memory environment becomes $s_1 = \{x \mapsto [1,1], y \mapsto \top\}$.

The process of computing s_1 based on s_0 is represented by the transfer function of x = 1. Additionally, the join is defined as $[l_1, u_1] \sqcup [l_2, u_2] = [min(l_1, l_2), max(u_1, u_2)]$. The partial-order relation is defined as $[l_1, u_1] \sqsubseteq [l_2, u_2]$ if and only if $l_1 \geq l_2$ and $u_1 \leq u_2$. For example $[1,3] \sqcup [7,10] = [1,10]$ and $[4,6] \sqsubseteq [1,10]$.

We use $\mathbb{S}$ to denote the set of all memory environments. $\mathbb{S}$ is a lattice with properly defined top ($\top$) and bottom ($\bot$) elements, join ($\sqcup$), partial-order ($\sqsubseteq$), and a widening operator [13]. Each node $n \in \mathbb{N}$ has a *transfer function* $t \in \mathbb{S} \longrightarrow \mathbb{S}$, taking an environment $s' \in \mathbb{S}$ as input (before executing the atomic operation in n) and returns a new environment $s \in \mathbb{S}$ as output.

Let $\mathsf{TFunc} \in \mathbb{N} \longrightarrow (\mathbb{S} \longrightarrow \mathbb{S})$ be a map from each node to its transfer function. For example, given a node $n \in \mathbb{N}$ whose operation is x = a+1, if $s' = \{x \mapsto [1,3], a \mapsto [2,5]\}$, the new environment is $s = (\mathsf{TFunc}(n))(s') = \{x \mapsto [3,6], a \mapsto [2,5]\}$.

The goal of an abstract interpreter is to compute an environment map $M \in (\mathbb{N} \longrightarrow \mathbb{S})$ over-approximating the memory state at every program location. M, typically, initially maps all variables in the entry node to $\top$, and all variables in other nodes to $\bot$. Then, it iteratively applies the transfer function $\mathsf{TFunc}(n)$ and joins the resulting environments for all n, until they reach a fixed point.

Without getting into more details (refer to the literature [13]), we define the sequential analyzer as a fixed-point computation with respect to the function $\mathsf{AnalyzeSeq} \in (\mathbb{N} \longrightarrow \mathbb{S}) \longrightarrow (\mathbb{N} \longrightarrow \mathbb{S})$:

$$\mathsf{AnalyzeSeq}(M) = n \mapsto (\mathsf{TFunc}(n))(\bigsqcup_{(n',n)\in\delta} M(n'))$$

Here, $M(n')$ is the environment produced by a predecessor node n' of n, and $s' = \bigsqcup_{(n',n)\in\delta} M(n')$ is the join of these environments. $(\mathsf{TFunc}(n))(s')$ is the new memory environment produced by executing the operation in n. Applying this function to all nodes of a sequential program until a fixed point leads to an over-approximated memory state for each program location.

However, directly applying the sequential analyzer to each execution of a multithreaded program is not practical because it leads to an exponential complexity. Instead, thread-modular techniques [40, 47–49] iteratively apply AnalyzeSeq to each thread, as if it were a sequential program, and then merge/propagate the global memory effects across threads. The iterative process continues until memory environments in all threads stabilize.

Since each thread is analyzed in isolation, this approach is more scalable than non-thread modular techniques. However, it may result in accuracy loss because the analyzer for each thread relies on a coarse-grained abstraction of *interferences* from other threads. When analyzing a thread t in the presence of a set of threads T, for example, the interferences are the effects of global memory stores from all $t' \in T$. The interferences are a map $(\mathbb{V} \longrightarrow 2^{\mathbb{S}})$ from each variable $v \in \mathbb{V}$ read by thread t to the set of memory environments

[1] For ease of presentation we assume a variable maps to a single value. Our analysis can trivially use relational domains.

produced by interfering stores, where $\mathbb{V}$ is the set of all program variables, and $2^{\mathbb{S}}$ is the power set of $\mathbb{S}$.

Prior thread-modular techniques [47–49] eagerly join all interfering memory states from the other threads in T before propagating them to the current thread t. As such, they often introduce bogus *store-to-load* data flows into the static analysis or miss valid *store-to-load* data flows. In the remainder of this paper, we present our method for mitigating this problem.

4 DECIDING INTERFERENCE FEASIBILITY

In this section, we describe our new method for quickly deciding the feasibility of a combination of store-to-load data-flows under a given memory model. An *interference combination* is a set $ic = \{(l, s), \ldots\}$ where each $(l, s) \in ic$ is a load l and an interfering store s.

Checking the feasibility of ic is formulated as a deductive analysis with inputs: (1) the flow graph of the current thread, (2) the flow graphs of all interfering threads, and (3) the existing set of store-to-load data flows represented by $\{(l, s) | (l, s) \in \textsc{ReadsFrom}\}$. The output of this deductive analysis is the relation MustNotReadFrom. $(l, s) \in \textsc{MustNotReadFrom}$ means the load l must not read from the store s since our analysis proved the data flow from s to l is infeasible given the input ReadsFrom relation in ic.

Consider the program in Figure 3 as an example. One thread interference combination we want to check is the load of `y` from `y=10` and the load of `x` from the initial value 0. Let these load and store instructions be denoted l_y, s_y^{10}, l_x, and s_x^0, respectively. Then, the feasibility problem is stated as follows: given $(l_y, s_y^{10}) \in \textsc{ReadsFrom}$, check if $(l_x, s_x^0) \in \textsc{MustNotReadFrom}$.

4.1 Notations

Before presenting the details of our feasibility checking procedure, we define a set of unary and binary relations over instructions and program variables. Specifically, $(s_1, v_1) \in \textsc{IsLoad}$ denotes s_1 is a load of variable v_1, and $(s_2, v_2) \in \textsc{IsStore}$ denotes s_2 is a store to variable v_2. We use $(s_1, \_) \in \textsc{IsLoad}$ if we do not care about the variable. Similarly, we use $f \in \textsc{IsFence}$ to denote that f is a fence. We also use IsLLMembar, IsLSMembar, IsSLMembar, IsSSMembar to denote load–load, load–store, store–load, and store–store memory barriers as defined in the SPARC architecture [65]; for example, a load–store `membar` prevents loads before the barrier from being reordered with subsequent stores.

We define binary relations over instructions s_1 and s_2: the first four relations (Dominates, NotReachableFrom, ThreadCreates, ThreadJoins) are determined by the program's flow graphs. Based on them, we deduce the MHB relation, which must be satisfied by all program executions. The ReadsFrom relation comes from the given ic, from which we deduce the MustNotReadFrom relation.

$(s_1, s_2) \in$	Dominates means that s_1 dominates s_2 in the control flow graph of a thread.
$(s_1, s_2) \in$	NotReachableFrom means that s_1 cannot be reached from s_2 in the control flow graph of a thread.
$(s_1, s_2) \in$	ThreadCreates means s_1 is the thread creation and s_2 is the first operation of the child thread.
$(s_1, s_2) \in$	ThreadJoins means s_1 is the thread join and s_2 is the last operation of the child thread.
$(s_1, s_2) \in$	MHB means that s_1 must happen before s_2 in all executions of the program.
$(s_1, s_2) \in$	ReadsFrom means that s_1 is a load that reads the value written by the store s_2.
$(s_1, s_2) \in$	MustNotReadFrom means that s_1 must not read from the value written by s_2.

Consider Figure 3 again, where we want to check if the load of `y` in the second thread reads from `y=10`, then is it possible for the load of `x` to read from the initial value `0`? In this case, we encode the assumption as $(l_y, s_y^{10}) \in \textsc{ReadsFrom}$. Next, we deduce the MustNotReadFrom relation. Finally, we check if $(l_x, s_x^0) \in \textsc{MustNotReadFrom}$.

4.2 Relaxing the Program-Order Requirement

To model the program order imposed by different memory models, we define a new relation NoReorder such that $(s_1, s_2) \in \textsc{NoReorder}$ if the reordering of s_1 and s_2 within the same thread is not allowed.

We define the rules for NoReorder based on the allowed program-order relaxations for different memory models (Figure 5).

For SC, NoReorder is defined as:

$$\frac{\top}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under SC)}$$

That is, no reordering is ever allowed under SC (row SC Figure 5).

For TSO, NoReorder is defined as:

$$\frac{(s_1, \_) \in \textsc{IsLoad}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under TSO)}$$

$$\frac{(s_2, \_) \in \textsc{IsStore}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under TSO)}$$

Under TSO, two operations (s_1, s_2) can not reorder in six of the eight cases. The first rule above disallows Columns 2, 3, 6, and 7 (Figure 5), while the second disallows Columns 3, 5, 7, and 9. Thus, reordering is permitted in two cases: Columns 4 and 8.

Although this is counter-intuitive, note that $W(v_1) \rightarrow R(v_1)$ (Column 4) may be reordered in our analysis under TSO (and PSO and RMO) for soundness: it permits *read-own-write-early* behaviors. We detail this shortly in Section 4.6.

For PSO, NoReorder is defined as:

$$\frac{(s_1, \_) \in \textsc{IsLoad}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under PSO)}$$

$$\frac{(s_1, v_1) \in \textsc{IsStore} \land (s_2, v_1) \in \textsc{IsStore}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under PSO)}$$

Under PSO, two operations (s_1, s_2) can not reorder in five of the eight cases. The first rule above disallows Columns 2, 3, 6, and 7, while the second disallows Column 5. Thus, reordering is permitted only in the remaining three cases (Columns 4, 8, and 9).

For RMO, the inference rules are defined as:

$$\frac{(s_1, v_1) \in \textsc{IsLoad} \land (s_2, v_1) \in \textsc{IsLoad}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under RMO)}$$

$$\frac{(s_1, v_1) \in \textsc{IsLoad} \land (s_2, v_1) \in \textsc{IsStore}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under RMO)}$$

$$\frac{(s_1, v_1) \in \textsc{IsStore} \land (s_2, v_1) \in \textsc{IsStore}}{(s_1, s_2) \in \textsc{NoReorder}} \text{ (under RMO)}$$

Similarly, the above inference rules can be directly translated from Columns 2, 3, and 6 of the table in Figure 5.

4.3 Handling Fences and Memory Barriers

Next, we present the ordering constraints imposed by fences and memory barriers. We consider four variants of the `membar` instruction, which prevents loads and/or stores before the `membar` from being reordered with subsequent loads and/or stores [65].

$$\frac{\begin{array}{c} m \in \textsc{IsLLMembar} \land (s_1, \_) \in \textsc{IsLoad} \land (s_2, \_) \in \textsc{IsLoad} \\ \land (s_1, m) \in \textsc{Dominates} \land (m, s_2) \in \textsc{Dominates} \end{array}}{(s_1, s_2) \in \textsc{NoReorder}}$$

$$\frac{(s, s_{sta}) \in \textsc{ThreadCreates}}{(s, s_{sta}) \in \text{MHB}} \qquad \frac{(s, s_{end}) \in \textsc{ThreadJoins}}{(s_{end}, s) \in \text{MHB}} \tag{1}$$

$$\frac{(s_1, s_2) \in \textsc{Dominates} \wedge (s_2, s_1) \in \textsc{NotReachableFrom} \wedge (s_1, s_2) \in \textsc{NoReorder}}{(s_1, s_2) \in \text{MHB}} \tag{2}$$

$$\frac{(s_1, s_2) \in \text{MHB} \wedge (s_2, s_3) \in \text{MHB}}{(s_1, s_3) \in \text{MHB}} \tag{3}$$

$$\frac{(l, s_1) \in \textsc{ReadsFrom} \wedge (s_1, s_2) \in \text{MHB} \wedge (l, v) \in \textsc{IsLoad} \wedge (s_1, v) \in \textsc{IsStore} \wedge (s_2, v) \in \textsc{IsStore}}{(l, s_2) \in \text{MHB}} \tag{4}$$

Figure 6: Deduction rules for MHB (must-happen-before).

$$\frac{(l, s) \in \text{MHB}}{(l, s) \in \textsc{MustNotReadFrom}} \tag{5}$$

$$\frac{(l_1, s_1) \in \textsc{ReadsFrom} \wedge (l_1, s_2) \in \text{MHB} \wedge (s_2, l_2) \in \text{MHB} \wedge (l_1, v) \in \textsc{IsLoad} \wedge (l_2, v) \in \textsc{IsLoad} \wedge (s_2, v) \in \textsc{IsStore}}{(l_2, s_1) \in \textsc{MustNotReadFrom}} \tag{6}$$

Figure 7: Deduction rules for the MustNotReadFrom.

$$\frac{m \in \textsc{IsLSMembar} \wedge (s_1, \_) \in \textsc{IsLoad} \wedge (s_2, \_) \in \textsc{IsStore} \wedge (s_1, m) \in \textsc{Dominates} \wedge (m, s_2) \in \textsc{Dominates}}{(s_1, s_2) \in \textsc{NoReorder}}$$

$$\frac{m \in \textsc{IsSLMembar} \wedge (s_1, \_) \in \textsc{IsStore} \wedge (s_2, \_) \in \textsc{IsLoad} \wedge (s_1, m) \in \textsc{Dominates} \wedge (m, s_2) \in \textsc{Dominates}}{(s_1, s_2) \in \textsc{NoReorder}}$$

$$\frac{m \in \textsc{IsSSMembar} \wedge (s_1, \_) \in \textsc{IsStore} \wedge (s_2, \_) \in \textsc{IsStore} \wedge (s_1, m) \in \textsc{Dominates} \wedge (m, s_2) \in \textsc{Dominates}}{(s_1, s_2) \in \textsc{NoReorder}}$$

We also model fences in terms of `membar`s since they prevent loads and stores from being reordered with subsequent loads and stores as well.

$$\frac{f \in \textsc{IsFence}}{f \in \textsc{IsLLMembar}} \qquad \frac{f \in \textsc{IsFence}}{f \in \textsc{IsLSMembar}}$$

$$\frac{f \in \textsc{IsFence}}{f \in \textsc{IsSLMembar}} \qquad \frac{f \in \textsc{IsFence}}{f \in \textsc{IsSSMembar}}$$

In addition to fences explicitly added to the program, there are fences implicitly added to thread routines such as `lock/unlock` and `signal/wait`. For example, in the code snippet `x = 1; lock(lk); a = y; unlock(lk)`, there is a fence inside `lock(lk)`, to ensure `x = 1` always takes effect before `a = y`. This is how most modern programming systems guarantee data-race-freedom [3] to application-level code (i.e., programs without data races have only SC behaviors). Thus, we model every call s_c to a POSIX thread routine using $s_c \in$ IsFence.

4.4 Rules for Deducing MustNotReadFrom

We divide our inference rules into two groups. The first (Figure 6) use the relations ThreadCreates, ThreadJoins, Dominates, and NoReorder to generate the must-happen-before (MHB) relation.

Rule (1) states that if the instruction s creates a thread with entry instruction s_{sta}, then s must happen before s_{sta}. Similarly, if instruction s joins a thread with exit instruction s_{end}, then s_{end} must happen before s.

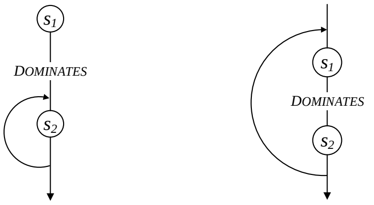

Figure 8: Example illustrating Rule (2)

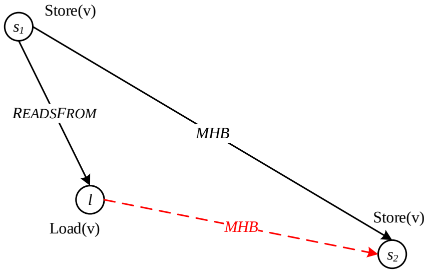

Figure 9: Example illustrating Rule (4)

Rule (2) states that if s_1 dominates s_2 within a thread's CFG, and s_1 is not reachable from s_2, (i.e., no loop encompasses both s_1 and s_2), then, if permitted by the memory model, s_1 must happen before s_2. Figure 8 exemplifies this rule: the loop in the left CFG is outside the Dominates edge, thus $(s_1, s_2) \in$ NotReachableFrom. The loop in the right CFG encompasses the Dominates edge, thus $(s_1, s_2) \notin$ NotReachableFrom.

Rule (3) states that the MHB relation is transitive: if s_1 must happen before s_2, and s_2 must happen before s_3, then s_1 must happen before s_3. Correctness follows from the definition of MHB.

Rule (4) states that if a load l reads from the value written by the store s_1, then l must happen before some second store to the same variable s_2 takes effect. This is intuitive because, if s_2 takes effect before l (but after the first store s_1), then l can no longer read from s_1. Figure 9 exemplifies this rule. Its correctness is obvious.

The second group of inference rules (Figure 7) takes the relations MHB and ReadsFrom and generates the MustNotReadFrom relation. Recall that if a load-store pair $(l, s) \in$ MustNotReadFrom, the value stored by s can never flow to l. Thus, MustNotReadFrom may be used to eliminate infeasible data flows.

Rule (5) states that if a load l must happen before a store s, then l cannot read from s. This follows from the definition of MHB. Note that a store s "happens" when it propagates to main memory.

Rule (6) states that if a load l_1 reads from a store s_1, and l_1 must happen before some other store s_2, and s_2 must happen before a second load l_2, then l_2 cannot read from s_1. Figure 10 exemplifies this rule. This is correct because l_2 reading from s_1 would mean s_1 takes effect after s_2 thus preventing l_1 from reading s_1.

4.5 Soundness and Incompleteness

When deciding the feasibility of an interference combination our analysis is designed to be sound but incomplete. By sound we mean it permits all possible program behaviors allowed by a memory model. Therefore, if it says a certain interference combination is

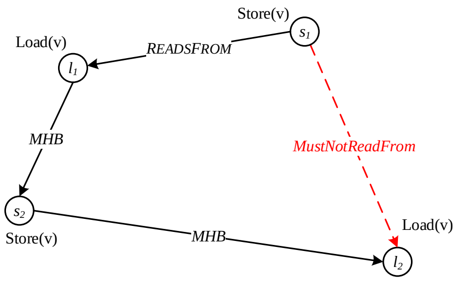

Figure 10: Example illustrating Rule (6).

infeasible it must be infeasible. However, there is no guarantee every infeasible interference combination will be found.

Incompleteness is expected: the intent is a quick pruning of infeasible combinations before the computationally expensive thread-level analysis. The overhead of insisting on completeness would outweigh its benefit: the feasibility checking problem, in the worst case, is as hard as program verification itself, which is undecidable.

Now, we formally state the soundness of our deductive procedure. First, our deduction of the NoReorder relation relaxing the program order requirement, from Figure 5, is sound.

Theorem 4.1. *Let s_1 and s_2 be two instructions in the same thread. If our rules deduce $(s_1, s_2) \in$ NoReorder, then the reordering of s_1 and s_2 is not allowed by the corresponding memory model.*

The proof of this theorem is straightforward, since our inference rules for deducing NoReorder directly follow the memory model semantics provided by Adve and Gharachorloo [4] in Figure 5.

Next, we note that, given the ReadsFrom relation, the deduction of the MustNotReadFrom relation is also sound.

Theorem 4.2. *Let l and s be two instructions. If our rules deduce to $(l, s) \in$ MustNotReadFrom, then l cannot read from s.*

The proof of this theorem is straightforward: it amounts to proofs of Rules (1)–(6). During the previous presentation, we have argued why each rule is correct. More formal proofs can be obtained via *proof-by-contradiction*, which is straightforward. We omit the details for brevity.

4.6 Relaxing the Write-Atomicity Requirement

Our method soundly models *buffer forwarding*, which corresponds to the write-atomicity requirement (Column 10 Figure 5). This allows a thread to read its own write before the written value is flushed to the memory, thus becoming visible to other threads. This is modeled in both the thread-level analyzer (AnalyzeSeq) and the deduction rules.

AnalyzeSeq captures the relaxation for free. During this analysis each thread is treated as a sequential program: all loads read their values from the preceding writes within the same thread.

The deduction rules for NoReorder (Section 4.2) always permit the reordering of a store with a subsequent load of the same variable (Column 4, Figure 5). That is, if $(s_1, v_1) \in$ IsStore and $(s_2, v_1) \in$ IsLoad, we do not deduce $(s_1, s_2) \in$ NoReorder due to buffer forwarding (even though it is counter-intuitive). Within a thread t, it may appear to be the case that the store and load are reordered from the perspective of all threads $t' \neq t$. Forbidding this reordering would be equivalent to forcing a full flush of the store-buffers before every load, thus prohibiting any thread from reading its own store earlier than other threads.

```
void thread1() {
  x = 1; // s1
  a = x; // s2
  b = y; // s3
}
```

```
void thread2() {
  y = 1; // s4
  fence; // s5
  c = x; // s6
}
```

```
assert( !(a ==1 && b == 0 && c == 0) );
```

Figure 11: Write atomicity example under TSO.

Figure 11 exemplifies the requirement of this relaxation. First, the assertion may be violated under TSO. An error trace is: `x = 1; a = 1; b = 0; y = 1; flush y; c = 0; flush x`. To permit this trace, we must allow the following interference combination: s_2 reads from s_1, s_3 reads from the initial value 0, and s_6 reads from the initial value 0. This combination is feasible only when we avoid enforcing the program order between s_1 and s_2. Specifically, the statements in thread 2 follow program order (s_4, s_5, s_6) from the fence. In thread 1, s_2 and s_3 are ordered since they are added to NoReorder under TSO. But, statements s_1 and s_2 are not added to NoReorder, thus preventing the assertion from being (incorrectly) verified.

5 THE THREAD-MODULAR ANALYSIS

Next, we present the integration of our interference analysis (Section 4) with a thread-modular analyzer. The thread-modular analyzer itself is standard, whose full details may be found in several prior works including [48, 49] and [40]. Thus, our presentation of the analyzer itself will be terse. Instead, we shall focus on our main contribution, which is adding the capability of deducing infeasible interference combinations for weak-memory models: our method is sound for not only SC but also TSO, PSO, and RMO. Prior techniques were either MM-oblivious, or sound only for SC.

Given a load l of v, the *interferences* on l, within the thread-modular analysis, are the environments after all stores to v from other threads. The function $\mathsf{n}(G)$ takes a graph as input and returns the nodes of the graph. The interferences on the loads in a thread G is the least fixed point of the function $\mathsf{Interfs}'$.

$$\mathsf{Interfs}'(G, M, I) = l \mapsto \{e\} \cup I(l)$$

$$\textbf{where } e \text{ is the environment after a remote store } s \notin \mathsf{n}(G)$$

$$\text{to the same variable as loaded by load } l \in \mathsf{n}(G)$$

$$\mathsf{Interfs}(G, M) = \mathsf{lfp}(\mathsf{Interfs}'(G, M), I_\perp)$$

We use $\mathsf{Interfs}'(G, M, I)$ as shorthand for $\mathsf{Interfs}'(G, M)(I)$, where $\mathsf{Interfs}'(G, M)$ is a partially-applied function, and use $I_\perp$ as the initial map from loads to interfering-environments, i.e., one mapping all nodes to $\{\perp\}$. lfp computes the least fixed point. $\mathsf{Interfs}'$ depends on the existence of M, a map from each program location, in all threads, to an environment. We show shortly that the computation of M and the interferences is done in a nested fixed point.

We refer to an *interference combination*, $ic \in (\mathbb{N} \mapsto \mathbb{S})$, as a map from a load l to the memory environment after a store instruction from which l reads. This differs slightly from the definition of Section 4 where it is defined as a set of load-stores pairs. The two can be easily converted as the analysis keeps track of all the environments associated with each store. Given the set of interferences I from Interfs, the set of all interference-combinations are all permutations of selecting a single environment from I for each load. The iterative thread-modular analysis separately considers each interference-combination thus increasing accuracy.

The *thread analyzer* adapts the sequential analyzer (AnalyzeSeq, Section 3) to use interference-combinations. AnalyzeTM′ takes a thread G and an interference combination ic and computes the input environment e for some node n in G by joining the environment after the predecessors of n with n's environment in ic, denoted $ic(n)$. Then, e is passed to n's transfer function to update $M(n)$.

$$\begin{aligned}
&\mathsf{AnalyzeTM}'(G, ic, M) = n \mapsto \mathsf{TFunc}(n)(e) \\
&\textbf{where } e = \bigsqcup_{(n', n) \in \mathsf{t}(G)} M(n') \sqcup ic(n) \\
&\mathsf{AnalyzeTM}(G, ic) = \mathsf{lfp}(\mathsf{AnalyzeTM}'(G, ic), M_\perp)
\end{aligned}$$

AnalyzeTM is the least fixed point of AnalyzeTM′. $\mathsf{t}(G)$ returns the transition-relation of a graph. $M_\perp$ is the initial memory map mapping the entry nodes of each thread to $\top$ and all others to $\perp$. Given a set of threads Gs and a set of interference-combinations I, applying AnalyzeTM to each $G \in Gs$ and each $ic \in I$ computes the analysis over all threads.

What remains is to show how the thread analyzer and the calculation of interferences can be done simultaneously since they are dependent: the interference computation depends on the analysis result, M, and the analysis result depends on the set of interferences, I. The solution is a nested fixed point: the outer computation produces M, and the inner computation produces I. The process iterates until M (and thus I) reach a fixed point.

$$\begin{aligned}
&\mathsf{Analyze}(G, M) = M' \\
&\textbf{where } I = \mathsf{Interfs}(G, M) \\
&\qquad\quad I' = \mathsf{FilterFeasible}(I) \\
&\qquad\quad M' = \mathsf{JoinMM}(\mathsf{map}(\mathsf{AnalyzeTM}(G), I')) \\
&\mathsf{AnalyzeAll}'(Gs, M) = \mathsf{JoinMM}(\mathsf{map}(\mathsf{Analyze}(M), Gs)) \\
&\mathsf{AnalyzeAll}(Gs) = \mathsf{lfp}(\mathsf{AnalyzeAll}'(Gs), M_\perp)
\end{aligned}$$

Analyze operates as follows: first, it takes M, the current analysis results over all threads, and computes the interferences, I, wrt the thread under test, G. The function FilterFeasible integrates the thread-level analyzer with the feasibility analysis of Section 4. It expands the interferences I into a set of interference combinations I', and filters any infeasible combination.

Specifically, given the interferences on a thread, $I = \{(l_1 \mapsto \{e_1, e_2, \ldots\}), (l_2 \mapsto \{e_3, e_4, \ldots\}) \ldots\}$, FilterFeasible creates all combinations of pairing each load to a single interfering environment, e.g., $I_e = \{\{\langle l_1, e_1\rangle, \langle l_2, e_3\rangle, \ldots\}, \{\langle l_1, e_2\rangle, \langle l_2, e_3\rangle, \ldots\}, \ldots\}$. Then, it maps each environment in I_e to the associated store generating the environment, e.g., $I_s = \{\{\langle l_1, s_1\rangle, \langle l_2, s_3\rangle\}, \ldots\}$. Each set of pairs of load and store statements in I_s is then passed to the deduction analysis of Section 4. If it is infeasible, it is discarded, otherwise it is added to the set I' returned by FilterFeasible.

$\mathsf{map}(\mathsf{AnalyzeTM}(G), I') \in 2^{(\mathbb{N} \mapsto \mathbb{S})}$ is the set of the results of applying $\mathsf{AnalyzeTM}(G, i)$ for each $i \in I'$. Specifically, $\mathsf{map} \in (A \to B) \to 2^A \to 2^B$ takes a function f and a set S, and returns a set containing the application of f on each element of S.

$\mathsf{JoinMM} \in (2^{(\mathbb{N} \mapsto \mathbb{S})} \to (\mathbb{N} \mapsto \mathbb{S}))$ takes the join of memory environments on matching nodes across a set of maps to join them into a single map. Similarly, AnalyzeAll′ joins the results of applying Analyze to the set Gs of threads. AnalyzeAll computes the fixed point of AnalyzeAll′ starting with the initial map $M_\perp$.

The following is a high-level example. Initially, each thread G is analyzed in the presence of $M_\perp$ resulting in the set of interferences, I, being empty (all stores map to $\perp$). The results of analyzing each thread are merged into a new map M. Each thread is then analyzed using M, resulting in the sets I and I' to be (potentially) non-empty, causing AnalyzeTM to be called once per-combination. Within a thread, the results of AnalyzeTM are joined, then, across threads, the results of Analyze are joined, creating M'. The procedure repeats, thus growing the size of M, I, and I' until $M = M'$.

We handle loops the same way as in prior techniques (e.g., [40]). Given a load l within a loop the previously described analysis can generate an infinite number of interference combinations for l, e.g., when l is within an infinite loop. Loops are unrolled when possible, and, when not, we join all the *feasible* interfering memory environments into a single value. An interfering environment e is infeasible to interfere on l if the store generating e must-happen after l; otherwise, it is feasible. This is sound for verifying assertions embedded in a concurrent program [40].

6 EXPERIMENTS

We implemented our weak-memory-aware abstract interpreter in a tool named FruitTree, building upon open-source platforms such as LLVM [5], Apron [37], and μZ [30]. Specifically, we use LLVM to translate C programs into LLVM bit-code, based on which we perform static analysis. We use the Apron library to manipulate abstract domains in the thread analyzer. We use the μZ fixed-point engine in Z3 [15] to solve Datalog constraints that encode the feasibility of interference combinations.

We implemented the state-of-the-art MM-oblivious abstract interpretation method of Miné [49], and the SC-specific method, Watts [40], on the same platform to facilitate experimental evaluation. We also compared against a previously implemented version of Duet [17, 18]. While Duet may be unsound, and Watts is unsound, we include their results because they are closely related to our new technique.

All methods implemented in FruitTree use the clustering and property-directed optimizations [40], where clustering considers interferences only within sets of loads, similar to the packing of relational domains, and property-direction filters interference combinations unrelated to properties under test. These optimizations reduce the number of interference combinations, which is crucial since it grows exponentially with respect to program's size.

We evaluated FruitTree on a large set of programs written using the POSIX threads. These benchmarks fall into two categories. The first are 209 litmus tests exposing non-SC behaviors under various processor-level memory models [8]. The second are 52 larger applications [17, 60, 61], including several Linux device drivers. The benchmarks total 61,981 lines of code. The properties under verification are assertions embedded in the program's source code: a property is valid if and only if the assertion holds over all executions under a given memory model.

Our experiments were designed to answer the following research questions: (1) Is our new method more effective than prior techniques in obtaining correctness proofs on relaxed memory? (2) Is our new method more accurate than prior techniques in detecting potential violations on relaxed memory? (3) Is our new method reasonably efficient when used as a static program analysis technique? We conducted all experiments on a Linux computer with 8 GB RAM, and a 2.60 GHz CPU.

6.1 Litmus Test Results

First, we present the litmus test results. Since these programs are small in terms of code size, all methods under evaluation (Miné, Watts, Duet, and FruitTree) finished quickly. Thus, our focus is not on comparing the runtime performance but comparing the accuracy of their results. Specifically, we compare our method to these state-of-the-art techniques in terms of the number of true proofs, bogus proofs, true alarms, and bogus alarms.

Here, a bogus alarm is a valid property which cannot be proved. A bogus proof is a property which may be violated yet is unsoundly and incorrectly proved. The litmus tests are particularly useful not only because they cover corner cases, but also because we know a priori if a property holds or not.

Table 1: Results on the litmus test programs under TSO.

Method	True Alarm	Bogus Alarm	True Proof	Bogus Proof	Time (s)
Miné [49]	77	207	8	0	12.9
Duet [17]	77	181	34*	0	473.1
Watts [40]	63	13	0	216★	71.0
FruitTree	77	72	143	0	89.2

Table 1 summarizes the litmus test results under TSO. The first column shows the name of each method, and the next four show the number of true alarms, bogus alarms, true proofs, and bogus proofs generated by each method, respectively. Since Watts [40] was designed to be SC-specific, it ignores non-SC behaviors, meaning its proofs are unsound under weaker memory (marked by ★). The last column is the total analysis time over all tests.

Overall, the results show the prior thread-modular technique of Miné admits many infeasible executions thus leading to 207 bogus alarms. Duet reported 181 bogus alarms. In contrast, our method (FruitTree) reported only 72 bogus alarms, together with 143 true proofs. Therefore, it is more accurate than these prior techniques.

Although Watts reported only 13 bogus alarms, it is unsound for TSO: it only considers SC behaviors and cannot be trusted. Furthermore, the soundness of Duet under TSO or any other non-SC memory model was not clear (since Duet was only designed for SC). Thus, in the result table, its 34 proofs are marked with *.

Table 2: Results on the litmus test programs under PSO.

Method	True Alarm	Bogus Alarm	True Proof	Bogus Proof	Time (s)
Miné [49]	81	203	8	0	12.9
Duet [17]	81	177	34*	0	473.1
Watts [40]	64	12	0	216★	71.0
FruitTree	81	68	143	0	281.4

Table 2 summarizes the results under PSO. Again, Watts may be unsound for weak memory. The same litmus programs were used under PSO as in TSO but the properties changed, i.e., whether an alarm is true or bogus. Note that Miné only verified 8 properties, Duet verified 34, whereas our method verified 143.

Table 3 summarizes the results under RMO. Under RMO, a different set of litmus programs were used since the instruction set for processors using RMO differs from TSO and PSO. Nevertheless, we observed similar results: FruitTree obtained significantly more true proofs and fewer bogus alarms than the other methods.

In general, our method was more accurate than prior techniques. However, since the analysis is over-approximated, it does not eliminate all bogus alarms. Currently, most bogus alarms reported by FruitTree require reasoning across more than two threads, e.g., the correctness of a property may require reasoning that thread T_1 reading $x = 1$ from thread T_2 implies $y = 1$ in thread T_3. Since our method is thread-modular—threads are analyzed individually by abstracting all other threads into a set of interferences—it cannot capture ordering constraints involving more than two threads. In principle, this limitation can be lifted by extending our interference feasibility analysis: we leave this as future work.

Table 3: Results on the litmus test programs under RMO.

Method	True Alarm	Bogus Alarm	True Proof	Bogus Proof	Time (s)
Miné [49]	28	67	8	0	4.9
Duet [17]	11	58	34	0	187.8
Watts [40]	0	0★	75	28★	33.9
FruitTree	28	13	62	0	46.9

6.2 Results on Larger Applications

Next, we present our results on the larger benchmark programs. Since execution time is no longer negligible, we compare, across methods, both the run time and accuracy. However, since the programs are larger (60K lines of code) and there are far too many properties to manually inspecting each case, we do not report the number of bogus alarms and bogus proofs due to lack of the ground truth. Instead, we compare the *total* number of proofs reported by each method, to show our method is more accurate even though all methods are approximate.

Table 4 shows our results under TSO, where ★ and * mark the unsoundly verified properties. Since the results for PSO and RMO are similar to Table 4, we omit them for brevity. Column 1 of this table shows the name of the benchmark program. Columns 2–3, 4–5, 6–7, and 8–9 show the run time and number of properties verified by Miné, Duet, Watts, and FruitTree, respectively.

Again, while the proofs reported by FruitTree and Miné's method are sound, the proofs reported by Watts are not, and the soundness of Duet on weak memory is unclear.

Overall, FruitTree proved 4,577 properties compared to only 1,712 proved by Miné, an increase of 2.7x more properties relative to prior state-of-the-art. Additionally, though Duet may be unsound, it proved only 2,432 properties. The definitely-unsound Watts "proved" 4,583 properties, possibly including bogus proofs.

In terms of the run time, FruitTree took 5,387 seconds, which is similar to Watts, and slower than Duet and Miné. However, the additional time is well justified due to the significant increase in the number of proofs. Furthermore, the runtime performance – proving 1 property per second – remains competitive as a static analysis technique.

To summarize, our new method has modest runtime overhead compared to prior techniques, but vastly improved accuracy in terms of the analysis results, and is provably sound in handling not only SC but also three other processor-level memory models.

7 RELATED WORK

We reviewed prior work on thread-modular abstract interpretation, which are either MM-oblivious [47–49] or SC-specific [40] in processor memory-models. There are also techniques [17, 18, 35, 53] that are not thread-modular.

There are code-transformation techniques [14, 39, 45] that transform a non-SC program into an SC program and then apply abstract

Table 4: Results on larger applications (total of 61,981 LOC).

	Minè [49]		Duet [17]		Watts [40]		FruitTree	
Name	Time	Verif	Time	Verif	Time	Verif	Time	Verif
thread00	0.01	0	1.1	0	0.03	0	0.03	0
threadcreate01	0.02	1	0.8	1	0.05	2	0.04	2
threadcreate02	0.02	1	0.7	2	0.03	2	0.03	2
sync_01_true	0.03	1	1.3	1	0.06	1	0.06	1
sync_02_true	0.03	1	1.2	1	0.07	1	0.07	1
intra01	0.02	1	1.2	0	0.04	2	0.04	2
dekker1	0.10	3	1.3	2	6.1	4	6.6	4
fk2012_v2	0.04	3	1.4	3	0.3	4	0.2	4
keybISR	0.04	4	1.2	4	2.1	6	2.3	5
ib700wdt_01	1.7	23	1.8	35	14.5	46	11.4	46
ib700wdt_02	13.2	63	2.7	95	108.4	126	89.4	126
ib700wdt_03	23.1	81	2.8	122	178.4	162	156.3	162
i8xx_tco_01	1.0	14	2.8	28	97.4	39	53.3	39
i8xx_tco_02	8.5	34	4.6	68	1288.3	99	757.3	99
i8xx_tco_03	10.7	37	4.7	74	1677.0	108	952.3	108
machzwd_01	0.6	14	1.3	14	107.5	35	42.1	34
machzwd_02	1.6	24	1.4	24	240.5	65	75.3	64
machzwd_03	4.1	39	1.8	39	488.7	110	128.9	109
mixcomwd_01	0.8	12	1.9	21	169.3	34	15.3	33
mixcomwd_02	2.2	32	4.4	41	768.0	64	61.7	63
mixcomwd_03	3.5	64	2.9	65	88.3	100	84.0	100
pcwd_01	0.7	10	0.9	22	1.8	31	1.4	31
pcwd_02	4.4	27	1.3	56	8.5	82	6.8	82
pcwd_03	10.2	40	1.7	82	18.5	121	15.4	121
pcwd_04	25.4	60	2.1	122	46.1	181	39.8	181
sbc60xxwdt_01	1.1	21	2.0	0	4.9	43	3.0	43
sbc60xxwdt_02	3.3	40	2.3	0	13.7	81	8.0	81
sbc60xxwdt_03	7.0	60	3.2	0	33.4	121	17.9	121
sc1200wdt_01	1.0	22	1.3	10	15.1	33	10.6	33
sc1200wdt_02	6.9	58	2.2	28	64.5	87	47.6	87
sc1200wdt_03	26.2	102	3.3	50	197.0	153	146.7	153
smsc37b787wdt_01	1.0	22	1.2	23	16.7	46	12.0	46
smsc37b787wdt_02	9.2	76	2.3	77	91.4	154	67.3	154
smsc37b787wdt_03	26.4	130	3.4	131	286.1	262	197.5	262
sc520wdt_01	1.5	15	1.1	16	15.6	45	11.0	45
sc520wdt_02	12.6	41	1.7	42	74.1	123	56.0	123
sc520wdt_03	27.8	58	2.2	59	155.6	174	116.9	174
w83877fwdt_01	12.5	34	1.8	34	83.9	137	71.0	137
w83877fwdt_02	29.0	50	2.3	50	189.6	201	159.6	201
w83877fwdt_03	54.2	66	2.7	66	357.9	265	301.9	265
wdt01	0.2	3	1.4	13	65.9	14	27.6	14
wdt02	0.3	5	1.5	21	600.2	22	306.5	22
wdt03	0.5	6	1.5	25	1479.1	26	766.2	26
wdt977_01	1.3	9	1.9	35	83.0	43	49.7	43
wdt977_02	2.4	13	2.3	51	115.6	63	76.0	63
wdt977_03	8.2	25	2.9	99	264.9	123	193.9	123
wdt_pci01	1.2	11	1.1	31	5.9	52	4.6	52
wdt_pci02	8.9	31	1.8	91	33.2	152	26.3	152
wdt_pci03	23.9	51	3.0	151	93.5	252	72.7	252
pcwd_pci_01	4.7	56	1.3	89	27.2	116	21.1	116
pcwd_pci_02	9.8	70	1.4	132	52.6	158	40.6	158
pcwd_pci_03	20.3	88	2.0	186	97.7	212	72.7	212
Total	415 s	**1752**	106 s	**2432***	9830 s	**4583**★	5387 s	**4577**

interpretation. They generally follow the *sequentialization* approach pioneered by Lal and Reps [41], with a focus on code transformation as opposed to abstract interpretation. To ensure termination, they make various assumptions to bound the program's behavior. Furthermore, they are not thread-modular, and often do not directly handle C code. Instead, they admit only *models* of concurrent programs written in artificial languages; because of this, we were not able to perform a direct experimental comparison.

In the context of bounded model checking, Alglave et al. proposed several methods for concurrent software on relaxed memory. They are based on either sequentializing concurrent programs [8] or encoding weak memory semantics using SAT/SMT solvers [7, 9]. Alglave et al. also developed techniques for modeling and testing weak-memory semantics of real processors [10], and characterized the memory models of some GPUs [6]. However, these techniques are primarily for detecting buggy behaviors as opposed to proving that such behaviors do not exist.

In the context of systematic testing, often based on stateless model checking [22, 25, 64] or predictive analysis [33, 54, 56, 57, 62, 63], a number of methods have been proposed to handle weak memory such as TSO/PSO [1, 16, 34, 67], PowerPC [2], and C++11 [51, 52]. However, since they rely on concretely executing the program, and require the user to provide test inputs, they can only be used to detect bugs. That is, since testing does not cover all program behaviors, if no bug is detected, these methods cannot obtain a correctness proof. In contrast, our method is based on abstract interpretation, which covers all possible program behaviors and therefore is geared toward obtaining correctness proofs.

Thread-modular analysis was also used in model checking [23, 24], where it was combined with predicate abstraction [29] to help mitigate state explosion and thus increase the scalability. However, model checking is significantly different from abstract interpretation in that each thread must be first abstracted into a finite-state model. Thread-modular analysis was also used to conduct shape analysis [26] and prove thread termination [12]. Hoenicke et al. [31] introduced a hierarchy of proof systems that leverage thread modularity in compositional verification on SC memory.

Similar to the interference analysis in Watts [40], we check the feasibility of thread interactions using Datalog. Datalog-based declarative program analysis was a framework introduced by Whaley and Lam [66]. Previously, it has been used to implement points-to [11, 42], dependency [28, 59] and change-impact analyses [27], uncover security bugs [44] and detect data races [50].

In abstract interpretation of sequential programs, Miné [46] proposed a technique for abstracting the global memory into a set of byte-level cells to support a variety of casts and union types. Ferrara et al. [20, 21] integrated heap abstraction and numerical abstraction during static analysis, where the heap is represented as disjunctions of points-to constraints based on values. Jeannet and Serwe [38] also proposed a method for abstracting the data and control portions of a call-stack for analyzing sequential programs with potentially infinite recursion. Subsequently, Jeannet [36] extended the work to handle concurrent programs as well. However, none of these methods was designed specifically for handling weak memory models.

8 CONCLUSIONS

We have presented a thread-modular static analysis method for concurrent programs under *weak memory models*, building upon a lightweight constraint system for quickly identifying the infeasibility of thread interference combinations, so they are skipped during the expensive abstract-interpretation based analysis. The constraint system is also general enough to handle a range of processor-level memory models. We have implemented the method and conducted experiments on a large number of benchmark programs. We showed the new method significantly outperformed three state-of-the-art techniques in terms of accuracy while maintaining only a moderate runtime overhead.

ACKNOWLEDGMENTS

This material is based upon research supported in part by the U.S. National Science Foundation under grants CNS-1405697 and CCF-1722710 and the U.S. Office of Naval Research under award number N00014-13-1-0527.

REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 353–367, 2015.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *International Conference on Computer Aided Verification*, pages 134–156, 2016.

[3] Sarita V. Adve and Hans-Juergen Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, 2010.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[5] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVM: A low-level virtual instruction set architecture. In *ACM/IEEE international symposium on Microarchitecture*, Dec 2003.

[6] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 577–591, 2015.

[7] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence. In *International Conference on Computer Aided Verification*, pages 508–524, 2014.

[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *European Symposium on Programming*, pages 512–532, 2013.

[9] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking ofÂăconcurrentÂăsoftware. In *International Conference on Computer Aided Verification*, pages 141–157, 2013.

[10] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 7, 2014.

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 243–262, 2009.

[12] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving thread termination. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–330, 2007.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[14] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 449–466, 2015.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[16] Brian Demsky and Patrick Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 20–36, 2015.

[17] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–308, 2012.

[18] Azadeh Farzan and Zachary Kincaid. Duet: Static analysis for unbounded parallelism. In *International Conference on Computer Aided Verification*, pages 191–196, 2013.

[19] Pietro Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *International Conference on Tests and Proofs*, pages 116–133. 2008.

[20] Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 302–321, 2014.

[21] Pietro Ferrara, Peter Müller, and Milos Novacek. Automatic inference of heap properties exploiting value domains. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 393–411, 2015.

[22] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[23] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, pages 262–277, 2002.

[24] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *International SPIN Workshop on Model Checking Software*, pages 213–224, 2003.

[25] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *International Conference on Computer Aided Verification*, pages 476–479, 1997.

[26] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 266–277, 2007.

[27] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference On Automated Software Engineering*, 2016.

[28] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[29] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *International Conference on Computer Aided Verification*, pages 262–274, 2003.

[30] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462, 2011.

[31] Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. Thread modularity at many levels: A pearl in compositional verification. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 473–485, 2017.

[32] Alan Huang. Maximally stateless model checking for concurrent bugs under relaxed memory models. In *International Conference on Software Engineering*, pages 686–688, 2016.

[33] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[34] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 447–461, 2016.

[35] Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for c#. In *International Symposium on Formal Methods*, pages 476–491, 2006.

[36] Bertrand Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2):285–306, 2012.

[37] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *International Conference on Computer Aided Verification*, pages 661–667. 2009.

[38] Bertrand Jeannet and Wendelin Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *International Conference on Algebraic Methodology and Software Technology*, pages 258–273, 2004.

[39] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–198, 2011.

[40] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[41] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[42] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–12, 2005.

[43] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[44] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[45] Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *International Symposium on Static Analysis*, pages 237–252, 2014.

[46] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, pages 54–63, 2006.

[47] Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Programming Languages and Systems*, pages 398–418. 2011.

[48] Antoine Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *Proc. of the 10th School of Modelling and Verifying Parallel Processes*, pages 35–48, 2012.

[49] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 39–58, 2014.

[50] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[51] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 131–150, 2013.

[52] Peizhao Ou and Brian Demsky. Checking concurrent data structures under the C/C++11 memory model. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–59, 2017.

[53] Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded java semantics for program verification. In *International Conference on Software Engineering*, pages 489–499, 2002.

[54] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

[55] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[56] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *Haifa Verification Conference*, pages 95–114, 2011.

[57] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *International Conference on Formal Methods and Models for Co-Design*, pages 99–108, 2011.

[58] Richard Sites. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[59] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static DOM event dependency analysis for testing web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[60] SVCOMP. International competition on software verification. `http://sv-comp.sosy-lab.org/2015/benchmarks.php`, Accessed: 2015-05-06.

[61] TLDP. Interrupt handlers: Linux kernel module programming guide. `http://www.tldp.org/LDP/lkmpg/2.6/html/x1256.html`, Accessed: 2015-05-06.

[62] Chao Wang and Malay Ganai. Predicting concurrency failures in generalized traces of x86 executables. In *International Conference on Runtime Verification*, pages 4–18, September 2011.

[63] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272, 2009.

[64] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[65] David L Weaver and Tom Gremond. *The SPARC architecture manual.* PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[66] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[67] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.