Constraint-Based Thread-Modular Abstract Interpretation

Markus J. Kusano

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Engineering

Chao Wang, Chair
Michael Hsiao, Co-Chair
Haibo Zeng
Patrick Schaumont
Dongyoon Lee

April 23, 2018
Blacksburg, Virginia

Keywords: concurrency, abstract interpretation, verification, numerical analysis, constraint solving, incremental analysis, weak memory

Copyright 2018, Markus J. Kusano
Constraint-Based Thread-Modular Abstract Interpretation

Markus Kusano

(ABSTRACT)

In this dissertation, I propose a set of constraint-based thread-modular abstract-interpretation techniques for static analysis of concurrent programs. Specifically, I integrate a lightweight constraint solver into a thread-modular abstract interpreter to reason about inter-thread interference more accurately. Then, I show how to extend this analyzer from programs running on sequentially consistent memory to programs running on weak memory. Finally, I show how to perform incremental abstract interpretation, with and without the previously mentioned constraint solver, by analyzing only regions of the program impacted by a program modification. I demonstrate, through experiments, that these new static analyzers are more accurate than prior abstract interpretation-based methods, with lower runtime overhead, and that the incremental technique can drastically speed up the analysis in the presence of small program

That this work received support from the Virginia Tech Bradley Fellowship, and the NSF’s Graduate Research Fellowship.
Software touches nearly every aspect of our lives, from smartphones, personal computers, and websites, to airplanes, cars, and medical equipment. Due to its ubiquity, we would like software in our lives to operate correctly, that is, without any unintended side effects, or freezes. This dissertation presents a new technique to automatically analyze a piece of software and determine if it runs as intended. We focus particularly on software where multiple entities run simultaneously, and thus can interact in many ways. Our automated analysis gives software developers high assurance that the software will always perform correctly, and thus never have any unexpected issues.

That this work received support from the Virginia Tech Bradley Fellowship, and the NSF’s Graduate Research Fellowship.
Acknowledgments

I would like to thank my advisor, Dr. Chao Wang, for his support throughout all the research underpinning this work: thank you for introducing me to this field, providing me guidance in both research and my career. Your help has been extremely beneficial in guiding my intellectual journey. I hope that you have enjoyed the process as much as I have.

Thank you to my committee co-chair, Dr. Michael Hsiao: I have always enjoyed your fun approach to both teaching and research; and thank you to the rest of my committee members Dr. Haibo Zeng, Dr. Patrick Schaumont, and Dr. Dongyoon Lee: without your time and feedback my degree would never have been completed.

My doctoral research was generously supported by funding from the National Science Foundation (NSF) Graduate Research Fellowship Program (GRFP) and the Virginia Tech Bradley Graduate Fellowship. Without their support I would have never had the opportunity to even start this work, let alone finish it.

Thank you to Albert, and Sasha for dissolution, trajectory, and ballast when I needed it most. And to my parents, John and Frau Doktor Gabriela Kusano, for their unquestioning support.

Finally, thank you to my wife, Bernadette, for always being a happy smiling person in my life, and guiding me to where I am today. Without you I would be vacillating.
# Contents

List of Figures vii  
List of Tables ix  

1 Introduction 1  
\hspace{1em} 1.1 Technical Abstract .................................................. 1  
\hspace{1em} 1.2 Concurrent Programs .................................................. 1  
\hspace{1em} 1.3 Background ............................................................... 3  
\hspace{1em} \hspace{1em} 1.3.1 Related Analysis Techniques ...................................... 7  
\hspace{1em} 1.4 Motivating Example ...................................................... 8  
\hspace{1em} 1.5 Constraint Based Abstract Interpretation ............................. 9  
\hspace{1em} 1.6 Handling Weak Memory .................................................. 10  
\hspace{1em} 1.7 An Incremental Analysis ................................................ 10  
\hspace{1em} 1.8 Contributions ............................................................. 11  
\hspace{1em} 1.9 Organization ............................................................... 12  

2 Constraint-Based Thread-Modular Abstract Interpretation 13  
\hspace{1em} 2.1 Motivation ................................................................. 17  
\hspace{1em} \hspace{1em} 2.1.1 What Is Thread-Modular Abstract Interpretation? .............. 17  
\hspace{1em} \hspace{1em} 2.1.2 Making the Analysis Flow-Sensitive ............................... 18  
\hspace{1em} \hspace{1em} 2.1.3 Program Representation: Control-Flow versus Data-Flow Graphs 19  
\hspace{1em} 2.2 Preliminaries .............................................................. 20
2.2.1 Sequential Abstract Interpretation ........................................ 20
2.2.2 Thread-Modular Abstract Interpretation ............................... 22
2.3 Flow-Sensitive Thread-Modular Analysis ................................. 23
  2.3.1 The New Algorithm ..................................................... 23
  2.3.2 Analyzing Interference Combinations In Isolation ................. 24
  2.3.3 Addressing Loops ..................................................... 26
  2.3.4 Soundness ............................................................. 27
2.4 Checking Interference Feasibility with Constraints ..................... 28
  2.4.1 The Program-Order and the Reads-From Constraints ............... 29
  2.4.2 Using Deduction to Check Interference Feasibility ................. 30
  2.4.3 The Running Example ................................................ 31
2.5 Clustering and Pruning Optimizations .................................... 32
  2.5.1 Property-Directing the Analysis .................................. 33
  2.5.2 Clustering Interferences via Dependencies ......................... 33
2.6 Experiments ................................................................... 34
2.7 Related Work .................................................................. 37
2.8 Discussion ..................................................................... 37

3 Thread-Modular Abstract Interpretation for Weak Memory .......... 41
  3.1 Motivating Examples ....................................................... 44
    3.1.1 The Example Under SC, TSO, PSO, and RMO ..................... 44
    3.1.2 Where Prior Techniques are Ineffective ........................... 45
    3.1.3 A New Way to Consider Weak Memory Models .................. 46
  3.2 Background .................................................................. 47
    3.2.1 Modeling Concurrency .............................................. 47
    3.2.2 Consistency of Memory Models .................................... 47
    3.2.3 Numerical Abstract Interpretation ................................. 48
  3.3 Constraint-Based Interference-Feasibility Checking for Relaxed-Memory Models 50
    3.3.1 Preliminaries .......................................................... 50
3.3.2 How the Program Order Constraint can be Relaxed . . . . . . . . . . 51
3.3.3 Modeling Memory Barriers and Fences . . . . . . . . . . . . . . . . 52
3.3.4 Deducing Interference Infeasibility . . . . . . . . . . . . . . . . . . 53
3.3.5 Proofs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.3.6 Allowing Writes to be Non-Atomic . . . . . . . . . . . . . . . . . . . 56
3.4 Analyzing Interferences Within Thread-Modular Abstract Interpretation . . 57
3.5 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
  3.5.1 Experiments on Litmus Tests . . . . . . . . . . . . . . . . . . . . . . 61
  3.5.2 Experiments on Larger Programs . . . . . . . . . . . . . . . . . . . . 62
3.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.7 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

4 Incremental Thread-Modular Abstract Interpretation 66
  4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
  4.2 Thread-Modular Abstract Interpretation . . . . . . . . . . . . . . . . . . 70
    4.2.1 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
    4.2.2 Thread-Modular Abstract Interpretation Without Constraints . . . 72
    4.2.3 Thread-Modular Abstract Interpretation With Constraints . . . . . 74
  4.3 Calculating Dependencies Under Weak Memory Models . . . . . . . . . . 79
    4.3.1 Proofs of Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . 81
  4.4 Change–Impact Analysis under Weak Memory Models . . . . . . . . . . . 82
  4.5 Incremental Thread-Modular Abstract Interpretation . . . . . . . . . . . 83
    4.5.1 Proofs of Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . 84
  4.6 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
  4.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
  4.8 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

5 Conclusions and Future Work 91
  5.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
### List of Figures

1.1 A tricky example for prior work. ........................................ 3
1.2 Numerical analysis of a sequential program. ....................... 4
1.3 A simple concurrent program for which there exists a rely-guarantee proof. 5

2.1 **WATTS**: Flow-sensitive thread-modular analysis. .............. 14
2.2 Proving that the **ERROR!** on $l_{13}$ is not reachable. .......... 15
2.3 Example: handling loops in thread-modular analysis. ............... 27
2.4 Deduction rules used by our feasibility analysis. .................. 30
2.5 Example: application of Rule 2.3. ...................................... 31
2.6 Example: application of Rule 2.6. ...................................... 31
2.7 Input and implied constraints for Figure 2.2. ....................... 32
2.8 Example: property directed redundancy pruning. ................... 33
2.9 The program dependence graph for Figure 2.8. ..................... 34
2.10 Example: dependency guided clustering. ............................. 34
2.11 Runtime overhead versus number of threads. ....................... 36

3.1 The assertion holds under SC, but not under TSO, PSO, and RMO memory models. .................................................. 42
3.2 **FRUIT TREE**: our memory-model-aware, thread-modular, abstract interpretation procedure. .................................... 43
3.3 The assertion always holds, but if the fence is removed, the assertion may fail under PSO and RMO. ............................ 44
3.4 Comparing the effectiveness of methods in handling the example programs in Figure 3.1 and Figure 3.3. ................................................................. 45

3.5 Allowed relaxations of various memory models (cf. [11]). $v_1$ and $v_2$ are distinct variables, and * indicates rule needs to be relaxed to allow read-own-write-early behaviors (Section 3.3.6). ........................................... 48

3.6 Deduction rules for MHB (must-happen-before). ......................................... 53

3.7 Deduction rules for the MUSTNOTREADFROM. ....................................... 53

3.8 Example for illustrating Rule (3.2) .......................................................... 54

3.9 Example illustrating Rule (2.3) ............................................................... 55

3.10 Example illustrating Rule (3.6). .............................................................. 55

3.11 Write atomicity example under TSO. ..................................................... 57

4.1 Overall flow of our incremental analysis. ................................................... 67

4.2 Message passing synchronization via fences. .............................................. 68

4.3 Modified version of thread one in Figure 4.2. .......................................... 69

4.4 Dependency analysis of Figure 4.2. .......................................................... 70

4.5 Example for the constraint-based abstract interpreter. ............................... 75

4.6 Dependency graph for Figure 4.5. ............................................................ 80
## List of Tables

2.1 Running prior approaches [52,119–121] on Figure 2.2. .................. 17  
2.2 Statistics of the benchmarks in our experiments. .................... 39  
2.3 Experimental results in the interval domain. ......................... 40  

3.1 Results on the litmus test programs under TSO. ...................... 61  
3.2 Results on the litmus test programs under PSO. ...................... 61  
3.3 Results on the litmus test programs under RMO. .................... 62  
3.4 Results on larger programs: * indicates unsoundly verified properties. .... 63  

4.1 Summary of benchmarks.................................................. 86  
4.2 TSO test results of constraint-based analysis ......................... 87  
4.3 Non-constraint analysis results........................................ 88
Chapter 1

Introduction

1.1 Concurrent Programs

Parallel processing hardware are now ubiquitous: they are in small consumer electronics—from cellphones to GameBoys to laptops—and in large intercontinental clusters and supercomputers. This ubiquity requires programmers to write concurrent software in order to make full use of their computing power. However, writing correct and efficient concurrent software is notoriously difficult: due to scheduling non-determinism, the number of possible program states can be astronomically large, quickly becoming far too large for a human to reason about. Furthermore, due again to scheduling non-determinism, traditional testing approaches, i.e. running the program with fixed inputs, is no longer effective in detecting subtle concurrency bugs that manifest only under an extremely small fraction of the feasible thread schedules. This difficulty requires automated analysis and verification tools, which can aid in the detection of bugs or proving the absence of these bugs in a concurrent program.

Software bugs in general, and concurrency bugs in particular, are becoming a real-world issue. For example, a concurrency bug in the firmware of a recent Intel processor [95] gave attackers access to write to the processor’s flash memory, allowing maleware to run independently to the operating system. Concurrency bugs exist also in numerous large-scale open-source software systems. For example, data-races in Mozilla’s Firefox web-browser [123] could potentially lead to security issues such as reading uninitialized memory [124], buffer overflows [125], and use-after-free bugs [126]. GCC’s [63] implementation of the C++ standard library (libstdc++ [64]) contained data-races, similarly causing buffer overflows [61] and use-after-free bugs [62] within its std::string implementation. Again, such bugs, while having the same consequence (e.g., a buffer overflow occurs) as bugs in sequential programs, are uniquely concurrent since they only manifest on an extremely small subset of all possible thread schedules. In general, fixing bugs takes up over 50% of the software development cost [23]. One reason why bug fixing is particularly expensive is because, in practice, 70% of
Figure 1.1: A tricky example for prior work.

the first version of bug patches end up being incorrect [145]. This has been shown to lead to product delays and significant financial losses [5].

Exacerbating these difficulties are the weak-memory models underlying modern computer architectures, such as total store ordering (TSO) [142, 161], partial store ordering (PSO) [148], and relaxed memory ordering (RMO) [161]. Weak memory models permit the processor’s reordering of instructions running within a thread, e.g., delaying expensive memory stores, thereby introducing more non-determinism into the program execution. As a result, a program may be proved correct, for example, under the sequential consistency (SC) memory model or TSO, while being buggy under PSO or RMO. For example, an issue like this occurred in the Postgresql database [132], where a bug existed only when the application runs on the Power [4] architecture. Such bugs cannot be detected by traditional testing approaches unless the software is thoroughly tested on all these hardware architectures. Thus, it would be beneficial for static verification tools to support various memory models and reason about their subtly differing behavior independently.

As a concrete example, consider the program in Figure 1.1 which exemplifies issues with prior static analysis techniques. We start by outlining the behavior of this program on a computer with sequential consistency (SC) [107] memory\(^1\). The first thread writes 5 to \(x\) and then sets \(flag\) to \(true\). The second thread reads the value of \(flag\), and if it is \(true\), reads the value of \(x\). The second thread then checks if the value of \(x\) is five, and, if not, reaches an error. Under SC, the program is free of reachability violations: the second thread reading \(flag\) as \(true\) implies that the write of 5 to \(x\) has already occurred\(^2\) meaning the value of \(x\) read by the second thread must be 5. However, existing thread-modular analysis techniques [119–121] are unable to prove the correctness of this program. This is because, at a high level, these techniques assume that, when a thread loads a value from the shared memory, the value it gets may be \(any\) value stored into this memory location by other threads.

While it is possible for non-thread-modular static analyzers [39, 86] to verify the property

\(^1\)SC is a memory model assuming that all statements within a thread execute in the order they are written in the program; or, equivalently, that all threads have a single consistent view of the shared memory.

\(^2\)Here, and in what follows, we use “occurred” to mean the value written to a variable was propagated to the shared memory and thus became visible to all threads.
in Figure 1.1, we consider their runtime overhead to be too significant, especially for larger programs. This is because, in the worst case, the computational overhead of these techniques increases exponentially with respect to the number of threads. Specifically, they maintain the control location of each thread explicitly, and thus if there are $m$ threads and $n$ control-locations in each thread, there are $n^m$ possible states. Thread-modular analyses are of interest in such cases particularly because they aim to make the overhead of analyzing a concurrent program close to that of analyzing a sequential program. As shown by experimental evaluations in prior work [121], as well as ours, this is often the case in practice.

1.2 Background

First, we briefly introduce static program analysis based on abstract interpretation [38]. For a full treatment, see, e.g., Neilson and Neilson [128]. Abstract interpretation is a technique capable of reasoning about all program executions without individually examining each execution. For example, consider the sequential program in Figure 1.2. The function cool_func takes two integers, $x$ and $y$, as input and returns an integer $ret$. The value of $ret$ depends on the values of $x$ and $y$: if $x$ is larger than 10 and $y$ is less than 300, $ret$ is 50; otherwise, $ret$ is 55. While this example is contrived, and the assertion may be verified using simpler techniques\(^3\), we include it to show how abstract interpretation can collapse multiple executions into a single abstract execution.

A concrete execution, i.e., executing/interpreting it faithfully as C code, of cool_func considers values of $x$ and $y$ (e.g., 15 and 0, respectively) and returns the correspondingly calculated value of $ret$ (50). Abstract interpretation, on the other hand, represents multiple concrete executions into a single abstract execution. For example, instead of considering individual values for $x$ and $y$, we could consider them as intervals [37], e.g., $x = [0, 5]$ and $y = [10, 20]$, meaning that $x$ could be any value between 0 and 5, and $y$ could be any value between 10 and 20. Such an abstraction combines together the 50 concrete executions where $x$ takes on 5 possible values, and $y$ takes on 10 possible values.

Generally, we could consider the value of $x$ and $y$ to be unbounded, i.e., their value falls in the interval $[-\infty, \infty]$ and then analyze cool_func. By doing so, we can reason about properties of the function valid over all possible values of $x$ and $y$. When entering the first branch, we know $x = [11, \infty]$ and $y = [-\infty, 299]$ due to the conditional statement guarding the branch ($x > 10 \land \land y < 300$). The update $ret = 50$ lets us know $ret = [50, 50]$. Similarly, in the second branch we have $x = [-\infty, 10]$ and $y = [300, \infty]$, and $ret$ is updated to $[55, 55]$. Once both branches have been analyzed, the results within each branch are joined together, giving us the fact that $x$ and $y$ are both $[-\infty, \infty]$ and, more importantly, that $ret = [50, 55]$. This value for $ret$ is sufficiently accurate to verify that $ret$ is never equal to 10 on all possible

\(^3\)For example, we can see that only constants (50 and 55) are stored into $ret$ so it cannot be anything but 50 and 55 and thus cannot be equal to 10. This, of course, does not generalize to arbitrary programs.
Markus Kusano  Chapter 1. Introduction  4

```c
int cool_func(int x, int y) {
    int ret;
    if (x > 10 && y < 300) {
        ret = 50;
    } else {
        ret = 55;
    }
    assert(ret != 10);
    return ret;
}
```

Figure 1.2: Numerical analysis of a sequential program.

```c
int x = 0;
void thread1() {
    x = 5;
}
```

```c
void thread2() {
    int t1 = x;
    assert(x < 10);
}
```

Figure 1.3: A simple concurrent program for which there exists a rely-guarantee proof.

The benefits of abstract interpretation versus traditional testing (applying specific inputs for \( x \) and \( y \) while running the program) is already apparent in this example. If we assume 64-bit integers then \( x \) and \( y \) may take on \( 2^{64} \) possible values each, and thus there are \( 2^{128} \) possible pairs of inputs for \( \text{cool} \_\text{func} \). Testing each individual input on a 2.8 GHz x86 machine would take about \( 2 \times 10^{12} \) years, i.e., longer than the estimated age of the universe [1]. In contrast, using abstract interpretation, the property can be verified for all possible inputs in less than a second.

Concurrent programs introduce non-determinism due to the scheduler selecting which thread to execute next, as well as the memory model permitting the instructions within a thread to be non-deterministically reordered. *Thread-modular* abstract interpretation [119–121] efficiently analyzes a concurrent program by analyzing each of its threads in isolation and then propagating the inter-thread effects, i.e., writes to the shared memory, across threads. In this way, the thread-modular analyzer avoids the upfront exponential blowup (exponential in the number of threads) of analyzing the entire program.

In some sense, thread-modular abstract interpretation can be viewed as automated technique for generating *rely-guarantee*-style proofs [89]. At a high level, the structure of a rely-guarantee proof enables reasoning about individual components (threads) in isolation, thus preventing the need for constructing a monolithic proof, which requires simultaneously reasoning about all threads.

\footnote{Assume that approximately \( 6.1 \times 10^{19} \) executions of \texttt{cool} \_\texttt{func} can be performed per second}
As a concrete example, consider the program in Figure 1.3 where the two threads share the variable $x$, initially set to 0. Thread one updates the value of $x$ to be 5; and thread two reads the value of $x$ into $t1$, and checks if $x$ is less than 10.

A monolithic (or non-thread-modular) analysis would consider each concurrent control state separately. Specifically, note that thread one can execute in two control states (before and after line 3), and thread two can execute in three control states (before and after line 2, and after line 3). Consider thread one’s control states as $s_{1,1}$ and $s_{1,2}$ and thread two’s control states as $s_{2,1}$, $s_{2,2}$ and $s_{2,3}$. Concretely, there are $2 \times 3 = 6$ possible global control states, one for each combination of control-location for thread one and thread two, that is:

1. $\langle s_{1,1}, s_{2,1} \rangle$,
2. $\langle s_{1,1}, s_{2,2} \rangle$,
3. $\langle s_{1,1}, s_{2,3} \rangle$,
4. $\langle s_{1,2}, s_{2,1} \rangle$,
5. $\langle s_{1,2}, s_{2,2} \rangle$,
6. $\langle s_{1,2}, s_{2,3} \rangle$.

Each of these global control states, conceptually, represents the execution of thread one simultaneously with thread two: while thread one is executing, thread two may be in any of its possible control states, and similarly for thread two. In general, such a monolithic analysis is exponential with respect to the number of threads: if we have $n$ threads each with $c$ control-locations, then the number of global control states is $c^n$, i.e., all combinations of control locations for each thread.

The thread-modular analysis, however, avoids explicitly enumerating all possible global control states. Instead, it analyzes each thread in isolation, as if it were a sequential program, but in the presence of an abstraction of all other threads in the program. These threads are abstracted as follows: instead of being an entity which moves through various control locations, each thread is modeled as something which stores values into shared memory. These stores are assumed to happen at arbitrary times. This abstraction is known as a thread interference (or simply interference). Conceptually, the interference enables a store from some thread to happen at any time. This means subsequently analyzing some other thread in the presence of this interference causes any load to see all interfering stores to be visible.

This can be seen concretely in the example program in Figure 1.3. If we analyze thread one as if it were a sequential program, we can see that it stores the value of 5 into $x$. Thus, we can abstract thread one as an interference of 5 onto $x$. Thread two, on the other hand, does not store into shared memory, so it can be abstracted into an interference which does nothing. Analyzing thread one as a sequential program causes the assertion to not be violated since the value of $x$ is 0 which is less than 10. Then, we can analyze thread two in the presence of the interference from thread one. This means that, when thread two loads $x$, it can see either the value local to the thread (the initial value of 0), or the value from the interference (5). Both 0 and 5 are less than 10, so the assertion in thread two again is not violated.
However, the increase in efficiency of the thread-modular analysis comes at a price: the analysis often causes many false alarms, i.e., exclaiming that a bug exists in the program when in fact the program is bug-free. We will examine such a case shortly. The reason why existing thread-modular techniques produce many false alarms is because they, essentially, permit all instruction reorderings across threads, even many that are infeasible. What is desired, then, is a technique to automatically (and cheaply) remove such infeasible reorderings, thereby increasing accuracy.

Related to the introduction of false alarms in the thread-modular analysis is that these thread-modular techniques are sound under weak-memory only by virtue of permitting all instruction reorderings: memory models are not considered independently (e.g., analyzing the program under TSO versus PSO) and, again, often produces false alarms. Specifically, the over-approximation of threads into an interference, essentially, permits all orderings of loads and stores across threads. This greatly over-approximates many memory models which restricts possible load–store orderings.

Additionally, programmers explicitly restrict load–store orderings using synchronization primitives (e.g., fences, memory barriers, locks, and condition variables) or implicit synchronization via control and data dependencies. Non thread-modular abstract interpreters [39] can analyze weak memory, but are impractical for real world programs, due to the exponential increase in complexity of analyzing the entire composed concurrent program.

### 1.2.1 Related Analysis Techniques

Next, we provide a high-level comparison between various existing analysis techniques related to numerical abstract interpretation. Largely, these techniques can be lumped into two categories: techniques geared toward bug-hunting, and techniques geared toward verification.

Bug-hunting techniques include symbolic execution [31, 36, 67, 98, 141], bounded model-checking [25, 35, 83, 157, 165], and gray-box fuzzing [2, 3, 68]. These techniques typically start from the initial state of the program and explore the state space. If during the exploration any error is reached, then a bug has been found in the program. These techniques cannot be used to verify the lack of errors in a program unless the program happens to have a finite number of reachable states and all these states have been explored. This differs from the topic of this dissertation, abstract interpretation, which instead aims at verifying the program (i.e., proving the absence of bugs) even if it has an infinite number of states.

Bug-hunting techniques such as fuzzing and symbolic execution are similar to traditional software testing techniques where test inputs are either manually crafted or randomly generated to achieve high coverage. Although originally developed for sequential programs, these techniques have been extended to concurrent programs as well. Specifically, there exists concurrent version of symbolic execution [34, 46, 70, 71], bounded model checking [55, 133, 146, 160], including bounded model checking for weak memory [15–18], stateless model
checking [127, 156, 158, 159, 164], including stateless model checking for weak memory [6, 7, 9, 41, 82, 99, 129, 171], and coverage-guided heuristic testing for concurrent programs [30, 45, 90, 112, 166–168].

The primary challenge in adapting bug-hunting techniques to concurrent programs is identifying the equivalent (or symmetric) executions. For example, if \( x = 5 \) runs in parallel to \( y = 10 \), there exists two possible executions: \( x = 5; y = 10 \) and \( y = 10; x = 5 \). However, since the writes to \( x \) and \( y \) are independent, the two executions, in the end, actually end up to the same final state. Identifying these independent orderings has the potential to significantly reduce the number of possible concurrent executions. Automatically identifying these redundancies are often achieved using partial-order reduction (POR) techniques [17, 58, 65, 69, 91, 92, 155, 160, 171].

Numerical abstract interpretation operates in a way similar to traditional data-flow analyses used in compilers, often called “static analyses,” such as the points-to analysis [149]. The main difference is that numerical abstract interpretation uses a numerical abstract domain, e.g., states mapping variables to their potential values, as opposed to a non-numerical domain, e.g., states mapping variables to the set of pointers they may point-to.

Verification techniques, which include simple control-flow analysis, data-flow analysis [60], abstract-interpretation [38], interpolation-based model checking [115, 116], k-induction [144], IC3 [27, 28], and property directed reachability [78], over-approximate the state space of the program in order to prove that no error is reachable. Compared to bug-hunting techniques, these methods aim at generating proofs for programs even with an unbounded state space, i.e., programs with infinite control-states (e.g., due to loops or recursive function calls) or infinite data-states (e.g., programs accepting arbitrary inputs). However, it is worth noting that most of these verification techniques, e.g., interpolation, k-induction, IC3, and property directed reachability, differ from abstract interpretation in that they use SAT/SMT based symbolic representations of the program state as opposed to a numerical abstract domain such as intervals [38] or octagons [118].

Miné [119–121] developed a number of static analyzers for concurrent programs based on thread-modular abstract interpretation. As mentioned in the previous example, such techniques automate the generation of a rely-guarantee-style proof within the given numerical abstract domain. Thread-modular analysis as a paradigm [56, 57, 77, 80] was used in many different contexts [24, 47, 49, 50, 69, 73–75, 84, 94, 113, 122, 139, 140]. Again, in general, it is used to automate the process of finding rely-guarantee proofs.

One difference between these techniques and Miné is the underlying representation of data in the program (e.g., using a binary-decision diagrams or SAT/SMT formulas as opposed to numerical abstract domains), or the underlying representation of the ordering of statements in the program (e.g., analyzing control-flow, data-flow, or some mixture of the two). Another difference is the interference abstraction [147] applied to the thread. We have showed in a previous example how a thread is abstracted into an unordered set of values stored into shared memory. Various other abstractions exist, e.g., checking if variables are monotonically
increasing [121], or more precisely checking the feasibility inter-thread data flow [47].

In summary, techniques such as abstract interpretation, predicate abstraction, IC3, and property-directed reachability are over-approximated verification techniques. If they report a program as bug-free, it is guaranteed to be bug-free. However, if they report some errors, these errors may be false alarms, e.g., if they are introduced due to over-approximation. This dissertation focuses on refining such over-approximation, and thus removing the false alarms produced by thread-modular abstract interpretation. Techniques such as bounded-model checking, symbolic execution, and “traditional” testing techniques (where the user provides test inputs, runs the program, and then checks against the expected outputs) are under-approximated, or “bug-hunting” techniques. That is, they can find real bugs (no false alarms) but cannot prove the absence of bugs.

1.3 Motivating Example

Now, we return to the example in Figure 1.1 and show why existing static analyzers [119–121] cannot produce sufficiently accurate results. First, each thread is analyzed in isolation as if it were a sequential program. Within thread 1, the analysis writes 5 and true to x and flag, respectively; and thread 2 loads the value of flag to be its thread-local value, i.e., the initial value false, and thereby does not enter the branch.

Next, each thread is analyzed again in the presence of the values stored into shared memory by the other thread. Since thread 1 does not perform any memory reads, its behavior remains the same as before. But, when thread 2 loads the value of flag it may read either its thread-local value of false, or true stored by thread 1 (i.e., $flag = [0, 1]$ in the interval domain). Since flag may be true, the thread enters the branch. Similarly, upon reading x, thread 2 may either read the local value of 0, or the value 5 from thread 1 (i.e., $x = [0, 5]$). Thus, the second branch can also be taken and the error state is reachable, causing a false alarm.

The reason why these existing techniques [119–121] cause a false alarm is because, when analyzing thread 2, the ordering of the statements within thread 1 is forgotten completely. On the one hand, forgetting the ordering of these statements allows the analysis to be very fast\(^5\) but, as shown in the example, often introduces false alarms.

\(^5\)Remembering the ordering of all statements withing thread 1 when analyzing thread 2 is equivalent to analyzing the concurrent program in a non-thread-modular way, i.e., analyzing the monolithic concurrent program.
1.4 Constraint Based Abstract Interpretation

The first portion of this dissertation aims to tackle the problem of reintroducing accuracy into the thread-modular abstract interpreter. We do this by integrating into the abstract interpreter an analysis automatically proving certain inter-thread data-flows as infeasible. We now exemplify this technique.

Reexamining the analysis of thread 2 in Figure 1.1, we can see there are four possible combinations of values read into flag and x:

- \( \rho_1 = \langle \text{flag} = 0 \land x = 5 \rangle \)
- \( \rho_2 = \langle \text{flag} = 0 \land x = 0 \rangle \)
- \( \rho_3 = \langle \text{flag} = 1 \land x = 5 \rangle \)
- \( \rho_4 = \langle \text{flag} = 1 \land x = 0 \rangle \)

If we look at how each combination affects thread 2, we can see that \( \rho_1 \) and \( \rho_2 \) do not enter the first branch in thread 1, so they do not cause the error to be reachable; \( \rho_3 \) enters the first branch but the value of x is 5, so the second branch is not taken. Only \( \rho_4 \) causes the error by entering both the first and second branches of thread 2. What we would like, then, since the error state in reality cannot be reached, is an automated way to prove \( \rho_4 \) infeasible.

The intuition behind this automatic reasoning on infeasibility is to construct a proof by contradiction: we assume the inter-thread data-flows did occur, and then automatically deduce the program-order implications of this assumption. If we reach a contradiction, then we know the combination is infeasible.

Concretely, consider \( \rho_4 \). Assuming the data-flow occurred we know:

- Thread 2 reads flag as true on line 9;
- so the write by thread 1 of true to flag must have occurred (line 6);
- since the statements in thread 1 execute in order, the write to x of 5 (line 4) must have occurred.
- We have reached a contradiction: the value of x cannot be 0.

This proof guarantees the infeasibility of \( \rho_4 \). Since the remaining combinations do not cause the error to be reachable, we have proved the program correct.

We formulate the infeasibility analysis as the solution of a system of constraints, specifically Datalog/Horn clauses in finite domains. Constructing the problem in this way ensures the constraint system can be solved in polynomial time—as opposed to exponential time as in SAT/SMT based techniques—thereby improving efficiency.
1.5 Handling Weak Memory

The infeasibility analysis of \( \rho_4 \) we discussed so far assume sequentially consistent memory. In reality, however, most of the computers have weaker memory models. Under a weaker memory model, the order of writes to \( x \) and \( \text{flag} \) in thread 1, for example, may be reversed by the processor's hardware, thereby causing the error to be reachable. Specifically, the program in Figure 1.1 is only correct if the write of \text{true} to \( \text{flag} \) occurs after the write of 5 to \( x \). That is, if thread 1 could potentially be rewritten as \( \text{flag} = \text{true}; x = 5 \), then the program would be incorrect. Modern computer architectures (e.g., TSO [142,161], PSO [148], RMO [161], IBM Power [4], ARM) and compiler optimizations [96] may, however, reorder such statements. As such, the programmer must include \textit{weak-memory primitives}, i.e., fences and memory barriers, into the program specifically forbidding such reordering.

Within our thread modular analysis, we handle weak memory by first relaxing the \textit{program-order} constraints assuming various weak memory models (TSO, PSO, and RMO in our work) and also introduce program-order constraints from weak-memory primitives. This makes our analysis sound under weak-memory and, of particular benefit, tailors our analyzer specifically for TSO, PSO, or RMO: existing techniques [121], for example, can not disambiguate between the three and thus often produces false alarms.

1.6 An Incremental Analysis

One technique to reduce analysis overhead is to make an analysis incremental. An incremental analysis takes an old, previously verified program, and a new unverified version, and then only analyzes the regions of the new program impacted by the change; intuitively, since the old version is already verified, only the new impacted regions need to be considered. Incremental analysis perfectly matches the typical nature of software development: small localized changes impacting only a small portion of the program.

However, there is no algorithm for incremental thread-modular abstract interpretation: when analyzing a new program version \( P \) using abstract interpretation, all statements in \( P \) are re-analyzed, even if only a small subset of \( P \) was impacted by the change. As such, particularly for large programs, the runtime overhead can be large.

The problem of designing an incremental abstract interpreter involves two major problems. First, a change-impact analysis [29,131,135,137] is required to identify statements in the program impacted by a modification. But, no existing change-impact analysis, to the best of our knowledge, targets multithreaded programs while handling weak memory. In other words, they are unable to answer questions such as “what statements in the program are impacted if I remove a particular memory fence?” We show how to integrate weak-memory primitives (such as fences) into a change-impact analysis, specifically by formulating their data-dependencies [96], thereby creating the first weak-memory aware change-impact analysis.
for concurrent programs. To keep the analysis scalable and efficient, we use what we call a *semi-flow-insensitive* analysis, where the majority of the program is analyzed flow-insensitively, but weak-memory related portions are analyzed flow-sensitively.

The second major problem in designing an incremental abstract interpreter is on integrating the change-impact information into the abstract interpreter. We show that the change-impact information can be used to both modify the transfer-functions of statements and preserve the prior analysis results across program versions. This combination of a weak-memory aware change-impact analysis, and its sound integration into the numerical abstract interpreter, create the first incremental thread-modular analyzer.

### 1.7 Contributions

We aim to solve the following problems:

1. How can thread-modular abstract interpretation be made more accurate?
2. How can thread-modular abstract interpretation be made aware of weak-memory models?
3. How can thread-modular abstract interpretation be made incremental?

First, in Chapter 2, we introduce the notion of a constraint-based thread-modular abstract interpreter: a lightweight constraint solver can be combined with a thread-modular abstract interpreter to reason about the infeasibility of concurrent executions. We show this can significantly reduce the number of false alarms when compared to prior work [119–121]. Additionally, the runtime overhead is minimal due to our proposed optimizations and the use of lightweight constraints solvable in polynomial time. Finally, the use of the constraint system still retains the benefits of abstract interpretation, namely its termination and soundness guarantee even when analyzing programs with infinite state.

Second, in Chapter 3, we show how this constraint-based thread-modular analyzer can be applied to programs running under weak memory. We introduce a further system of constraints specifically considering TSO, PSO, and RMO. This allows the tool to reason about these memory models, thereby making it capable to proving properties specific to each one. It is particularly useful for software running only under a specific memory model: behaviors of other memory models can be ignored. As a result, the tool becomes much more accurate when compared to verifiers not aware of memory models [119–121].

Third, we show how to make our thread-modular analysis incremental in Chapter 4. This involved two novelties: the first one is that we created and proved the correctness of our change-impact analysis for weak memory models, i.e., an analysis capable of identifying the impacted statements when moving from an old program version to a new one. This required us to define the semantics of program dependencies for weak-memory related operations. Existing change-impact analyses [70] only considered programs running on the sequentially
consistent memory. The second novelty is that we show how the change-impact analysis can be integrated into a thread-modular abstract interpreter. This significantly reduces the runtime of the analysis but still remains sound when analyzing programs running under weak memory models.

1.8 Organization

What remains is organized as follows: we introduce and prove the correctness of our constraint-based thread-modular abstract interpretation algorithm for sequentially consistent (SC) memory in Chapter 2. Then, we extend the constraint system from SC to TSO, PSO, and RMO in Chapter 3, thereby greatly increasing the accuracy relative to verifiers that are not aware of memory models at all. Finally, we present a change-impact analysis that is sound under weak memory models, and integrate it into the thread-modular abstract interpreter in Chapter 4. We obtain significant speedups by analyzing only the impacted regions of the program. Chapter 5 concludes and outlines visions of future work.
Chapter 2

Constraint-Based Thread-Modular Abstract Interpretation

Although abstract interpretation [37] has wide use in the analysis and verification of sequential programs, designing a scalable abstract-interpretation-based analysis for shared-memory concurrent programs remains a difficult task [47,52,119–121]. Due to the large concurrent state space, directly applying techniques designed for sequential abstract interpretation to interleaved executions of a concurrent program does not scale. In contrast, recent thread-modular techniques [52, 119–121] drastically over-approximate the interactions between threads, allowing a more tractable but less accurate analysis. Their main advantage is that sequential abstract interpreters can be lifted to concurrent ones with minimal effort. However, they consider thread interactions in a flow-insensitive manner: given a system of threads \{A,B,C\}, for instance, they assume A can observe all combinations of memory modifications from B and C despite that some of these combinations are infeasible, thereby leading to a large number of false alarms even for simple programs.

In this paper, we propose the first constraint-based flow-sensitive method for composing sequential abstract interpreters to form a more accurate thread-modular analysis. Though desirable, no existing static method is able to maintain inter-thread flow sensitivity with a reasonable cost. The main advantage of our new method is that, through the use of a lightweight system of constraints, it can achieve a high degree of flow sensitivity with negligible runtime cost. Here, our goal is to prove the correctness of a set of reachability properties of a program: the properties are embedded assertion statements whose error conditions are relational expressions over program variables at specific thread locations. Another advantage is that our method can be implemented as a flexible composition of existing sequential abstract-interpretation frameworks while retaining the well-known benefits of an abstract interpretation based analysis analysis, such as soundness and guaranteed termination as well as the freedom to plug in a large number of abstract domains [37,118].

Figure 2.1 shows an overview of our new method. Given an input concurrent program, our
method returns a set of relational and numerical invariants statically computed at each thread location as output. These invariants, in turn, can be used to prove the set of reachability properties of the program. During the thread-modular analysis, we first apply a sequential abstract interpreter to each individual thread and then propagate their results across threads before applying these sequential abstract interpreters again. The iterative process continues until a fix-point is reached over the set of invariants. During each iteration, the abstract interpreter also communicates with a Datalog engine to check if a thread interference, or set of interferences (data flow from global writes to reads), is feasible. If we can statically prove that the interference is infeasible, i.e., it cannot occur in any real execution of the program, we skip it, thereby reducing the analysis time and increasing accuracy.

In contrast to all existing methods in this domain, our thread-interference analysis is flow-sensitive for two reasons. First, we explore the memory interactions between threads individually by propagating their memory-states along data-flow edges without eagerly merging them through join operations as in prior techniques [52,119–121]. Second, we identify and remove the infeasible memory interactions by constructing and solving a system of lightweight happens-before causality constraints. These constraints (Horn clauses in finite domains) capture only the causality ordering of the program’s statements as opposed to the more complex relational and numerical properties. As such, they can be solved by a Datalog engine in polynomial time. These two techniques, together, can greatly reduce the number of false alarms caused by over-approximating the global memory state across threads, thereby allowing more programs and properties to be verified as correct compared to prior approaches.

Consider the program in Figure 2.2, which has two threads communicating through the shared variables \( x \) and \( \text{flag} \). Initially \( \text{flag} \) is false and \( x \) is 0. Thread 1 only performs shared memory writes by setting \( x \) to 4, and then 5, before setting \( \text{flag} \) to true. Thread 2 only performs shared memory reads: it reads the value of \( \text{flag} \) and if the value is true, reads the value of \( x \). Note that the \textbf{ERROR!} (at \( l_{13} \)) is unreachable since, for Thread 2 to reach \( l_{11} \), Thread 1 has to set \( \text{flag} \) to true (at \( l_6 \)) before \( l_9 \) is executed; but in such a case, \( l_5 \) must have

Figure 2.1: WATTS: Flow-sensitive thread-modular analysis.


Prior thread-modular analyzers such as Ferrara [52] and Miné [119–121] would have difficulty with this program because their treatment of inter-thread communication is flow-insensitive. That is, if one thread writes to a shared variable at program location $l_i$ and another thread reads the same shared variable at program location $l_j$, they would model the interaction by adding a data-flow edge from $l_i$ to $l_j$, even if the data-flow edge is infeasible or is only feasible in some program executions. For example, in Figure 2.2, no concrete program execution simultaneously allows the flow of $x$ from $l_4$ to $l_{11}$ and the flow of $\text{flag}$ from $l_6$ to $l_9$. In such cases, these prior methods would lose accuracy because their way of modeling the inter-thread data flow cannot differentiate between the feasible and infeasible data-flow combinations.

In contrast, our new method detects and eliminates such infeasible data-flows. For now, it suffices to say that our method would report that the flow of $x$ from $l_4$ to $l_{11}$ cannot co-exist with the flow of $\text{flag}$ from $l_6$ to $l_9$. We will provide full details of our constraint-based interference analysis in Section 2.4.

Our constraint-based method for checking the feasibility of inter-thread data flow is sound: when it declares a certain combination of interferences as infeasible, the combination is guaranteed to be infeasible in the actual program. However, note that for efficiency reasons, our method does not attempt to identify each and every infeasible combination. This is consistent with the fact that abstract interpretation, in the context of property verification, is generally an over-approximation: it is capable of proving the absence of errors but does not aim to guarantee that all unverified properties have real violations. As such, the additional effort we put into our constraint-based interference analysis is a fair trade-off between lower runtime overhead and improved accuracy. This puts our method in a nice middle ground between the more heavyweight model checkers [35] and the highly scalable and yet somewhat inaccurate static program analysis techniques [52, 119–121].

Another perhaps subtle benefit of our method is that the sequential abstract interpreter only needs a lightweight constraint solver [79] as a black-box to query the feasibility of a set of interferences. As such, it provides a flexible and extensible framework, allowing additional constraints, deduction rules, and decision procedures (e.g., solvers for symbolic-numerical domains) to be plugged in to further reduce the number of false alarms. To make our method more efficient, we also propose several optimizations to our interference feasibility
analysis (Section 2.5): we leverage control and data dependencies to group interferences before checking the feasibility of their combinations, and leverage property-directed pruning to reduce the program’s state space.

Our method also differs from the concurrent static analyzer developed by Farzan and Kincaid [47,48] despite that both methods employ a constraint-based interference analysis. The difference is due to the fact that we aim at solving a slightly different problem from theirs. First, our goal is to accurately analyze a concurrent program with a fixed number of threads, whereas their goal is to soundly approximate the behavior of parameterized programs (with an unbounded number of thread instances). Second, our method is strictly thread-modular: we iteratively apply a sequential abstract interpreter to a set of control-flow graphs, one per thread, and one at a time. In contrast, they analyze a single monolithic data-flow graph of the entire concurrent program. As a result, their method is significantly less accurate than ours on non-parameterized programs. We illustrate the difference between these two computational models, i.e., a set of per-thread control-flow graphs versus a monolithic data-flow graph, in Section 2.1.3.

We implemented our new method in a static analysis tool named Watts, for verifying reachability properties of multithreaded C/C++ programs written using the POSIX thread library. The tool builds upon the LLVM compiler, using the μZ [79] fix-point engine in Z3 [40] to solve Datalog constraints and the Apron library [87] to implement sequential abstract interpreters over numerical abstract domains. We have evaluated our method on a set of benchmarks with a total of 26,309 lines of code. Our experiments show that Watts can successfully prove 1,078 reachability properties, compared to 38 properties proved by the prior, flow-insensitive methods. Furthermore, Watts achieved the 28x increase in the number of verified properties with only a 1.4x increase in the analysis time.

In summary, this paper makes the following contributions:

1. We propose the first constraint-based flow-sensitive method for composing thread-modular abstract interpreters into a more accurate static analysis.
2. We develop a lightweight constraint-based framework for soundly checking the feasibility of inter-thread interferences and combinations of interferences.
3. We develop optimization techniques to improve the efficiency of our analysis by leveraging control and data dependencies and property-directed pruning.
4. We implement and evaluate our method on a large set of benchmarks to demonstrate its advantages over prior works.

The remainder of this paper is organized as follows. We use examples in Section 2.1 to illustrate the main ideas behind our method. Then, we provide a brief overview of sequential and thread-modular abstract interpretation in Section 2.2. We present our new algorithm in Sections 2.3, 2.4, and 2.5, and our experimental results in Section 2.6. We review related work in Section 2.7, and conclude in Section 2.8.
Table 2.1: Running prior approaches \([52, 119–121]\) on Figure 2.2.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interference</td>
<td>Interference</td>
</tr>
<tr>
<td>One</td>
<td>flag = {1} (x = {4, 5})</td>
<td>9,10</td>
</tr>
<tr>
<td>Two</td>
<td>flag = {1} (x = {4, 5})</td>
<td>9,10,11</td>
</tr>
</tbody>
</table>

### 2.1 Motivation

We present a series of examples showing applications of our new method compared to existing approaches.

#### 2.1.1 What Is Thread-Modular Abstract Interpretation?

First, Figure 2.2 to provide an overview of prior works on thread-modular abstract interpretation \([52, 119–121]\). These methods all use the same notion of interference between threads: an interference is a value stored into shared memory at some point during the execution of a thread. In Figure 2.2, there are three interferences, all from Thread 1: the writes to \(x\) at \(l_4\) and \(l_5\) and the write to \(\text{flag}\) at \(l_6\).

These prior techniques analyze the program by statically computing the over-approximated set of interferences for each thread:

1. Initially, the set of interferences in each thread is empty.
2. Each thread is independently analyzed in the presence of interferences from all other threads.
3. The set of interferences in each thread is recomputed based on the results of the analysis in Step 2.
4. Steps 2–3 are repeated until the interferences stabilize.

During the thread-modular analysis (step 2), each thread keeps track of its own memory environment at every thread location. The memory environment is an abstract state mapping program variables to their values. To incorporate inter-thread effects, when a thread performs a shared memory read on some global variable \(q\), it reads either the values of \(q\) in its own memory environment, or the values of \(q\) from the interferences of all other threads. These techniques rely on a flow-insensitive analysis in that each read may see all values ever written by any other thread, even if the flow of data is not feasible in all, or any, of the concrete program executions.

Table 2.1 shows the results of analyzing Figure 2.2 with prior thread-modular approaches \([52,\)
Column 1 shows the two iterations. Columns 2 and 4 show the lines reachable after each iteration in the two threads. Columns 3 and 5 show the interferences generated after each iteration. In the second iteration, the interferences generated during the first iteration are visible: Thread 2 is analyzed in the presence of the interferences generated by Thread 1. After two iterations, the interferences stabilize, which concludes the analysis. Unfortunately, the result in Table 2.1 shows that Thread 2 can reach \( l_{12} \), where it reads the value of \( x \) either from its own memory environment (the initial value 0) or from the interference of Thread 1 (4), thereby allowing the \texttt{ERROR!} to be reached. This is a false alarm: the property violation is generated because the inter-thread interferences are handled in a flow-insensitive manner.

In this example, to eliminate the false alarm one has to maintain a complex invariant such as \((\text{flag} = \text{true}) \rightarrow (x = 5)\) which cannot be expressed precisely as a relational invariant even in expensive numerical domains such as convex polyhedra. Additionally, in order to propagate such a relational invariant across threads, as in [121], they need to hold over all states within a thread. Otherwise, interference propagation is inherent non-relational. Specifically, propagating the interferences on a variable \( x \) first requires a projection on \( x \), thus forgetting all relational invariants.

In contrast, our method can eliminate the false alarm even while staying in inexpensive abstract domains such as intervals. In particular, our work shows that eagerly joining over all interference across threads is inaccurate and should be avoided as much as possible.

### 2.1.2 Making the Analysis Flow-Sensitive

We propose, instead, to partition the set of interferences from other threads into clusters and then consider combinations of interferences only within these clusters. In this way, we effectively delay the join of interferences and avoid the inaccuracies caused by eagerly joining in existing methods. For example, if we assume the three interferences in Figure 2.2 fall into one cluster (worst for efficiency but best for accuracy), our analysis of the program would be as follows: in the first iteration, we apply per thread abstract interpretation and then compute the interferences for each thread; these computations remain the same as in the first iteration of Table 2.1. In the second iteration, however, when analyzing Thread 2 at the point of reading \texttt{flag}, there will be six possible cases, due to the Cartesian product of \( x = \{0, 4, 5\} \) and \( \text{flag} = \{0, 1\} \).

Unlike prior approaches, which eagerly join these cases to form \( x = \{0, 4, 5\} \land \text{flag} = \{1, 0\} \), we analyze the impact of each case \( \rho_1 - \rho_6 \) individually as follows:

- \( \rho_1 \), corresponding to \( (x = 4 \land \text{flag} = 0) \);
- \( \rho_2 \), corresponding to \( (x = 5 \land \text{flag} = 0) \);
- \( \rho_3 \), corresponding to \( (x = 0 \land \text{flag} = 0) \);
- \( \rho_4 \), corresponding to \( (x = 5 \land \text{flag} = 1) \);
- \( \rho_5 \), corresponding to \( (x = 4 \land \text{flag} = 1) \); and
• $\rho_6$, corresponding to \((x = 0 \land \text{flag} = 1)\).

This leads to enough accuracy to prove the \textbf{ERROR!} is not reachable. First, when $\text{flag} = 0$ ($\rho_1$, $\rho_2$, and $\rho_3$) the \textbf{ERROR!} cannot be reached since the branch at $l_{10}$ will not be taken ($b_1$ is false). Second, in the case of $\rho_4$, the first branch at $l_{10}$ will be taken but the branch guarding \textbf{ERROR!} will not, since $x = 5$, meaning $t_1$ is also 5. For the two remaining cases ($\rho_5$ and $\rho_6$) our constraint-based interference analysis (Section 2.4) would show it is impossible to have both $x = 4 \land \text{flag} = 1$, or $x = 0 \land \text{flag} = 1$.

The intuition behind the analysis is that infeasible data flows cause a contradiction between program-order constraints and data-flow edges. Specifically, examining $\rho_5$, if Line 9 reads $\text{flag}$ as 1 and Line 11 reads $x$ as 4, then:

- Line 6 is executed before Line 9 ($b_1 == \text{true}$),
- Line 9 is executed before Line 11 (program order),
- Line 11 is executed before Line 5 ($t_1 == 4$), and
- Line 5 is executed before Line 6 (program order).

This leads to a contradiction since the above must-happen-before relationship forms a cycle, meaning the combination cannot happen. Similarly, $\rho_6$ is infeasible since the write of 1 to $\text{flag}$ implies the updates to $x$ have already occurred, meaning $x$’s initial value, 0, is not visible to Thread 2. At this point, the only feasible interferences do not cause an \textbf{ERROR!} — the program is verified.

To obtain the aforementioned accuracy with minimal computational overhead, we leverage the statically computed control and data dependencies to partition the set of interferences into clusters. This can significantly reduce the number of cases considered during our thread-modular analysis. For example, when a load of $y$ is independent of the subsequent load of $x$, e.g., the value loaded from $y$ has no effect on the load of $x$, the thread would have two unconnected subgraphs in its program dependence graph [51]. Unconnected subgraphs create a natural partition of loads into clusters, thereby significantly reducing the complexity of our interference-feasibility checking. This is because we only need to consider combinations of interferences \textit{within} each subgraph. We will show details of this optimization in Section 2.5.

### 2.1.3 Program Representation: Control-Flow versus Data-Flow Graphs

Our method also differs from DUET, a concurrent static analyzer for parametric programs developed by Farzan and Kincaid [47,48]. Although DUET also employs a constraint-based interference analysis, the verification problem it is designed to solve is significantly different from ours. First, their method is designed for soundly analyzing parameterized concurrent programs where each thread routine may have an unbounded number of instances. In contrast, our method is designed to analyze programs with a fixed number of threads with the goal of
obtaining more accurate analysis results.

Second, their method relies on running an abstract interpreter over a single monolithic data-flow graph of the entire program, whereas our method relies on running abstract interpreters over a set of thread-local control-flow graphs. The difference between using a set of thread-local control-flow graphs and a single monolithic data-flow graph can be illustrated by the following two-threaded program: 

\[
\begin{align*}
\{x++;\} & \text{ and } \{\text{tmp}=x;\}.
\end{align*}
\]

In the monolithic data-flow graph representation [47, 48], there would be cyclic data-flow edges between the read and write of \(x\) across threads as well as an edge from the write of \(x\) to itself. As a result, applying a standard abstract interpretation based analysis would lead to the inclusion of \(\text{tmp} = \infty\) as a possible value, despite that in any concrete execution of the program, the end result is either \(\text{tmp} = 1\) or \(\text{tmp} = 0\) (assume that \(x = 0\) initially). Our new method, in contrast, can correctly handle this program.

\section{2.2 Preliminaries}

We provide a brief review of abstract interpretation based static analysis for sequential and concurrent programs. For a thorough treatment, refer to Nielson and Nielson [128] and Miné [119–121].

\subsection{2.2.1 Sequential Abstract Interpretation}

An abstract interpretation based static analysis is a fix-point computation in some abstract domain over a program’s control-flow graph (CFG). The control-flow graph consists of nodes representing program statements and edges indicating transfer of control between nodes. Due to their one-to-one mapping we interchangeably use the term statement and node. We assume the graph has a unique entry.

The analysis is parameterized by an abstract domain defining the representation of environments in the program. An environment is an abstract representation of the memory state. The purpose of restricting the representation of memory states to an abstract domain is to reduce computational overhead and guarantee termination. For example, in the interval domain [37], each variable has an upper and lower bound. For a program with two variables \(x\) and \(y\), an example environment at some program location is \(x = [0, 5] \land y = [10, 20]\). With properly defined meet (\(\sqcap\)) and join (\(\sqcup\)) operators, a partial order (\(\sqsubseteq\)), as well as the top (\(\top\)) and bottom (\(\bot\)) elements, the set of all possible environments in the program forms a lattice. In the interval domain, for example, we have \([0, 5] \sqcup [10, 20] = [0, 20]\) and \([0, 5] \sqsubseteq [0, 2]\).

Each statement in the program is associated with a transfer function, taking an environment as input and returning a new environment as output. The transfer function of statement \(st\) for some input environment \(e\) returns a new environment \(e'\), which is the result of applying
Algorithm 1 Sequential abstract interpretation.
1: function SEQ_ABSINT(G : the control-flow graph)
2: \( \text{Env}(n) \) is initialized to \( \top \) if \( n \in \text{ENTRY}(G) \), else to \( \bot \)
3: \( WL \leftarrow \text{ENTRY}(G) \)
4: while \( \exists n \in WL \)
5: \( WL \leftarrow WL \setminus \{n\} \)
6: \( e \leftarrow \text{TRANSFER}(n, \text{Env}(n)) \)
7: for all \( n' \in \text{SUCCS}(G, n) \) such that \( e \not\sqsubseteq \text{Env}(n') \)
8: \( \text{Env}(n') \leftarrow \text{Env}(n') \sqcup e \)
9: \( WL \leftarrow WL \cup \{n'\} \)
10: return \( \text{Env} \)

\( st \) in \( e \). Consider the above example of interval domain for \( x \) and \( y \) again. The result of executing the statement \( x = x + y \) in the above example environment would be the new environment \( x = [10, 25] \land y = [10, 20] \).

For brevity, we will not define all the transfer functions for a programming language explicitly since the main contributions of this work are language-agnostic. As an example, however, consider the statement \( t = \text{load} \ x \), which copies a value from memory to a variable. Its transfer function can be represented as \( \lambda e.e[t = x] \), where \( e[st] \) is the result of evaluating \( st \) in the environment \( e \). Conceptually, it takes an input environment and returns a new environment where \( t \) is assigned the current value of \( x \).

The standard work-list implementation of an abstract-interpretation based analysis \cite{128} is shown in Algorithm 1. The input is a control-flow graph \( G \), where \( \text{ENTRY}(G) \) is the entry node and \( \text{SUCCS}(G, n) \) is the set of successors of node \( n \). \( \text{Env} \) is a function mapping each node \( n \) to an environment immediately before \( n \) is executed. The initial environment \( \top \) associated with the entry node means that all program variables can take arbitrary values, e.g., \( x = y = \cdots = [-\infty, \infty] \) for integer variables. The initial environments for all other nodes are set to \( \bot \) (the absence of values).

The work-list, \( WL \), is initially populated only with the entry node of the control-flow graph. The fix-point computation in Algorithm 1 is performed in the while-loop: a node \( n \in WL \) is removed and has its transfer function executed, resulting in the new environment \( e \). The function \( \text{TRANSFER} \) takes a node \( n \) and the environment \( \text{Env}(n) \) as input and returns the new environment \( e \) (result of executing \( n \) in \( \text{Env}(n) \)) as output. If a successor of the node \( n \) has a current environment with less information than \( e \) (as determined by \( \sqsubseteq \)), then it is added to the work-list and its environment is expanded to include the new information (Lines 7-9). The process proceeds until the work-list is empty, i.e., all the environments have stabilized. Standard widening and narrowing operators \cite{37} may be used at Line 8 to guarantee termination and ensure speedy convergence.
Algorithm 2: Thread-modular abstract interpretation.

1: function ThreadModAbsInt( Gs : the set of CFGs )
2: TE ← ∅
3: I ← ∅
4: repeat
5: I′ ← I
6: for all g ∈ Gs
7: i ← \{e | e ∈ I(g′), g′ ∈ Gs, and g′ ≠ g\}
8: Env ← SeqAbsInt-Modified(g, i)
9: TE ← TE⩫Env
10: for all (n, e) ∈ TE
11: if n is a shared memory write in g ∈ Gs
12: I(g) ← I(g)⩫Transfer(n, e)
13: until I = I′
14: return TE

2.2.2 Thread-Modular Abstract Interpretation

Next, we review thread-modular abstract interpretation: an iterative application of a sequential abstract interpreter on each thread in the presence of a joined set of interferences from all other threads. Since a thread-modular analysis never constructs the product graph of all threads in the program, it avoids the state space explosion encountered by non-thread-modular methods [86].

First, we make a slight modification to the previously described sequential abstract interpretation (Algorithm 1); the per-thread abstract interpretation must consider both the thread-local environment and the interferences from other threads. Here, an interference is an environment resulting from executing a shared memory write. Let SeqAbsInt-Modified(G, i) be the modified abstract analyzer, which takes an additional environment i as input. The environment i represents a joined set of interferences from all the other threads. We also modify the transfer function Transfer(n, Env(n)) of shared memory read as follows: for \(t=\text{load } x\), where x is a shared variable, we allow t to read either from the thread-local environment Env(n) or from i, the interference parameter. For example, if the thread-local environment before the load statement contains \(x = [10, 15]\) and the interference parameter contains \(x = [50, 60]\), we would have \(t = [10, 15]⩫[50, 60] = [10, 60]\).

Algorithm 2 shows the thread-modular analysis procedure. The input is the set Gs of control-flow graphs, one per thread. The output, TE, is a function mapping the thread nodes (nodes in all threads) to environments. During the analysis, each thread-local CFG g has an associated interference environment I(g): the environment is the join of all environments produced by shared memory writes in the thread g. Due to their one-to-one correspondence, we will use thread and its (control-flow) graph interchangeably.

Inside the thread-modular analysis procedure, both TE and I are initially empty. Then, the sequential abstract interpretation procedure is invoked to analyze each thread \(g ∈ Gs\).
2.3 Flow-Sensitive Thread-Modular Analysis

In this section, we present our new method for flow-sensitive thread-modular analysis. For ease of comprehension, we shall postpone the presentation of the constraint-based feasibility checking until Section 2.4, while focusing on explaining our method for maintaining inter-thread flow-sensitivity during thread-modular analysis.

2.3.1 The New Algorithm

Before diving into the new algorithm, notice that the reason why Algorithm 2 is flow-insensitive is because all environments from interfering stores of other threads are joined (Line 7) prior to the thread-modular analysis. Furthermore, within the thread-modular analysis routine, SEQABSTRACT-MODIFIED, the combined interfering environment, \( i \), is joined again with the thread-local environment during the application of the transfer function at each CFG node. Such eager join operations are the main sources of inaccuracy in existing methods. First, inaccuracy arises from the join operation itself: it tends to introduce additional behaviors, e.g., \([0,0] \sqcup [10,10] = [0,10]\). Second, a thread is allowed to see any combination of interfering stores even if some of them are obviously infeasible (e.g., Section 2.1, Figure 2.2).

To avoid such drastic losses in accuracy, we need to make fundamental changes to the thread-modular analysis procedure.

- For each thread \( g \in G_s \), instead of defining its interference as a single environment, we use a set of pairs \((n,e)\) where \( n \) is a CFG node of a shared memory write and \( e \) is the environment after \( n \).
- For each shared variable read, instead of it reading from the eagerly joined set of environ-
ments, we maintain a set, $LIs(l) = \{(n, e), \ldots\}$, where each $(n, e)$ represents an interfering store and the store’s interfering environment.

- For each thread $g \in Gs$, instead of representing the interferences from all other threads as the join of the interfering environments (Line 7, Algorithm 2), we represent them as a set $I_c$ of interference combinations: each $i_c \in I_c$ is a distinct combination of the store-to-load flows for all $l \in Loads(g)$.

Algorithm 3 shows our new analysis: in the remainder of this section, we shall compare it with Algorithm 2 and highlight their differences. There are two main differences. First, the interferences are represented as a set of pairs of store statements and their associated environment (Line 13). We modify $\sqcup$ to be the join of environments of pairs with matching nodes across two sets. Recall that if $A$ and $B$ are sets of pairs of the form $\{(n, e), \ldots\}$, then $A \sqcup B$ denotes the join of environments on the matching nodes. Second, we compute the set $I_c$ of feasible and non-redundant interference combinations (store-to-load flows) for a thread (Line 7) and analyze a thread in the presence of each combination individually (Lines 8–10). That is, for each call to the sequential abstract interpreter SeqAbsInt-Modified2, as the second parameter, instead of passing the join of interferences from all other threads, we pass each $i_c \in I_c$ to map every load to an interfering store individually.

### 2.3.2 Analyzing Interference Combinations In Isolation

Inside $\text{INTERFERENCECOMBOFEASIBLE}(g, I)$, we compute the set $I_c$ of feasible interference combinations. Here, $Loads(g)$ is the set of shared variable reads in thread $g$, $LoadVar(l)$ is the variable used in the load instruction $l$, and $StoreVar(s)$ is the variable stored-to in the store instruction $s$.

We first compute the set $VEs$ of interferences from other threads (Line 19); each pair $(n, e) \in VEs$ is a store and environment from a thread other than $g$. Then, we pair each load $l \in Loads(g)$ with any corresponding store in $VEs$ (Lines 20–29); the result is stored in $LIs$ which maps each load instruction $l$ to a set of stores in the form of $(n, e)$ pairs. The special pair $(s_{\text{dummy}}, e_{\text{self}})$ indicates the thread should read from its intra-thread environment. For now, ignore Lines 26–29 since they are related to the handling of loops — we discuss how loops are handled during the computation of interference combinations in the next subsection.

Next, the function $\text{CartesianProduct}$ takes $LIs$ as input and returns the complete set of interference combinations from $LIs(l_1) \times \cdots \times LIs(l_k)$. To make what we have explained so far clearer, consider an example program with two threads: $g_1$ and $g_2$. Thread $g_1$ has two loads, $Loads(g_1) = \{l_1, l_2\}$ such that $LoadVar(l_1) = x$ and $LoadVar(l_2) = y$. Thread $g_2$ has three interfering environments: two on $x$, $s_1$ and $s_2$, with associated environments $e_1$ and $e_2$, respectively; and another, $s_3$, on $y$, with environment $e_3$. Assume we are currently analyzing $g_1$ in the presence of interferences from $g_2$.

We first use the set $I$ of interferences to collect the interferences from $g_2$ in $VEs$, which
Algorithm 3 Flow-sensitive thread-modular analysis.

1: function THREADMODABSINT-FLOW(Gs: the set of CFGs)
2: \text{TE} \leftarrow \emptyset
3: \text{I} \leftarrow \emptyset
4: \text{repeat}
5: \quad \text{I}' \leftarrow \text{I}
6: \quad \text{for all } g \in \text{Gs}
7: \quad \quad \text{I}_c \leftarrow \text{INTERFERENCECOMBOFEASIBLE}(g, \text{I})
8: \quad \quad \text{for all } i_c \in \text{I}_c
9: \quad \quad \quad \text{Env} \leftarrow \text{SEQABSINT-MODIFIED}(g, i_c)
10: \quad \quad \text{TE} \leftarrow \text{TE} \sqcup \text{Env}
11: \quad \text{for all } (n, e) \in \text{TE}
12: \quad \quad \text{if } n \text{ is a shared memory write in } g \in \text{Gs}
13: \quad \quad \quad \text{I}(g) \leftarrow \text{I}(g) \sqcup \{\text{TRANSFER}(n, e)\}
14: \text{until } \text{I} = \text{I}'
15: \text{return } \text{TE}

16: function INTERFERENCECOMBOFEASIBLE(g, \text{I})
17: \text{I}_c \leftarrow \emptyset
18: \text{VES} \leftarrow \{(n, e) \mid (n, e) \in \text{I}(g'), g' \in \text{Gs}, \text{ and } g' \neq g\}
19: \text{for all } l \in \text{LOADS}(g)
20: \quad \text{LIs}(l) \leftarrow \{(s_{\text{dummy}}, e_{\text{self}})\}
21: \text{if } l \text{ is not self-reachable}
22: \quad \text{for all } (n, e) \in \text{VES}
23: \quad \quad \text{if LOADVAR}(l) = \text{STOREVAR}(n)
24: \quad \quad \quad \text{LIs}(l) \leftarrow \text{LIs}(l) \sqcup \{(n, e)\}
25: \text{else}
26: \quad \text{for all } (n, e) \in \text{VES}
27: \quad \quad \text{if } (\text{LOADVAR}(l) = \text{STOREVAR}(n))
28: \quad \quad \quad \text{\& } \neg \text{MUSTHAPPENBEFORE}(l, n)
29: \quad \quad \quad \text{LIs}(l) \leftarrow \text{LIs}(l) \sqcup \{(s_{\text{dummy}}, e)\}
30: \text{Es} \leftarrow \text{CartesianProduct}(\text{LIs})
31: \text{for all } i_c \in \text{Es}
32: \quad \text{if QUERY.ISFEASIBLE}(i_c)
33: \quad \quad \text{I}_c \leftarrow \text{I}_c \cup \{i_c\}
34: \text{return } \text{I}_c

is \{(s_1, e_1), (s_2, e_2), (s_3, e_3)\}$. Next, we compute \text{LIs} for the two loads \{l_1, l_2\} in thread $g_1$. We pair $l_1$ with the two interferences on $x$ from $s_1$ and $s_2$, and pair $l_2$ with the single interference on $y$ from $s_3$. Using $\cdots$ to denote a list of items, we represent the result as $\text{LIs}(l_1) = [(s_1, e_1), (s_2, e_2), (s_{\text{dummy}}, e_{\text{self}})]$ and $\text{LIs}(l_2) = [(s_3, e_3), (s_{\text{dummy}}, e_{\text{self}})]$. Without any optimizations, the resulting Cartesian product $\text{Es} = \text{LIs}(l_1) \times \text{LIs}(l_2)$ would contain the following items:

\[
\begin{align*}
\text{i}_{c_1} &= \{(l_1, (s_1, e_1)), (l_2, (s_3, e_3))\}, \\
\text{i}_{c_2} &= \{(l_1, (s_2, e_2)), (l_2, (s_3, e_3))\}, \\
\text{i}_{c_3} &= \{(l_1, (s_{\text{dummy}}, e_{\text{self}})), (l_2, (s_3, e_3))\}, \\
\text{i}_{c_4} &= \{(l_1, (s_1, e_1)), (l_2, (s_{\text{dummy}}, e_{\text{self}}))\}, \\
\text{i}_{c_5} &= \{(l_1, (s_2, e_2)), (l_2, (s_{\text{dummy}}, e_{\text{self}}))\}, \\
\text{i}_{c_6} &= \{(l_1, (s_{\text{dummy}}, e_{\text{self}})), (l_2, (s_{\text{dummy}}, e_{\text{self}}))\}.
\end{align*}
\]

For each combination $i_c \in \text{Es}$, we check if it is feasible (Lines 31–33): the infeasible combinations will be filtered out, and the result, $\text{I}_c$, is returned. We discuss how we determine the feasibility of an interference (Line 32) in Section 2.4.
Continuing with the algorithm’s description, on Line 9 the sequential abstract interpretation, \( \text{SEQABSINT-MODIFIED2} \), takes \( g \) and each \( i_c \in I_c \) as input and returns a node-to-environment map, \( \text{Env} \), as output. During this per-thread analysis, the transfer function of a load uses only \( i_c \) to determine the environment to use. When a load \( l_1 \) is being executed, if the special item \( \langle l_1, (s_{\text{dummy}}, e) \rangle \) is in \( i_c \), the load reads from its own thread-local environment at \( l_1 \); if the remote store environment \( \langle l_1, (s, e) \rangle \) is in \( i_c \), the load also reads from the remote environment \( e \).

At this point, we have improved the prior work (Algorithm 2) to avoid inaccuracies from over-approximations caused by the eager join over all interferences. The cost for this accuracy is explicitly testing each of the combinations of potential interferences. However, we have not presented our methods for clustering and pruning (Section 2.5) as well as checking if any of the combinations are \textit{infeasible} (Section 2.4). By applying such optimization techniques, we cannot only drastically reduce the overhead of running the abstract interpretation subroutine but also increase the accuracy.

### 2.3.3 Addressing Loops

In general, a load within a loop could execute many times. As a result, the number of stores that a load could read from is potentially infinite. To guarantee soundness and termination, we join all the interfering stores that \textit{may} affect a load in a loop with the environment within the thread at the time of the load. By doing this, we conservatively treat all these feasible interferences in a flow-insensitive manner for loads within loops.

Specifically, Lines 26–29 perform the join of interferences for loads within a loop. For a given load, all stores on the same variable that must-not-happen after the load are considered (we will further discuss the happens-before constraints in Section 2.4). For these conflicting stores, all of the environments are joined together on a single dummy node (\( s_{\text{dummy}} \)). In the end, each self-reachable load has a single (joined) environment. Consequently, during the Cartesian product computation, it will have a single interference. Within the sequential abstract interpreter, the load merges the thread-local environment and this single interfering environment.

However, even in such case, our new method is more accurate than the prior work. Consider the example in Figure 2.3. Thread 1 executes a load in a while-loop running an arbitrary number of times concurrently with thread 2 before creating thread 3. Because of the thread creation, there is a must-happen-before edge between the load in thread 1 (Line 5) and the write in thread 3 to \( x \). When constructing the interference combinations for the load in thread 1 (\( l \)), there are three potential stores: \( s_{10} \), \( s_{11} \), and \( s_{14} \) for the writes to \( x \) on Lines 10, 11, and 14, respectively.

When considering \( s_{10} \), the condition on Line 28 of our new algorithm is true since \( s_{10} \) does not always happen after \( l \) (and similarly for \( s_{11} \)). Therefore, \( LI(s(l)) \) is assigned \( \{s_{\text{dummy}}, e_{10}\} \)
int x = 0;
void thread1() {
  create(thread2);
  while (*) {
    int t1 = x;
  }
  create(thread3);
}

void thread2() {
  x = 1;
  x = 2;
}

void thread3() {
  x = 10;
}

Figure 2.3: Example: handling loops in thread-modular analysis.

initially, where $e_{10}$ is the environment at $s_{10}$. Next, $LI(s(l))$ is assigned $\{s_{dummy}, e_{10} \sqcup e_{11}\}$. Finally, for $s_{14}$, since it must happen after $l$, it is not added to $LI(s)$. When computing the Cartesian product, there is only a single load with a single location-store pair, so there is only one interference combination.

For this example, the analysis results in $t1$ being 0, 1, or 2. The value of 10 written by thread 3 is excluded using the must-happen-before constraint. So, although multiple interfering stores are merged for the single load within the loop, the accuracy of the analysis is still significantly higher than the entirely flow-insensitive analysis in prior works.

### 2.3.4 Soundness

Our new method in Algorithm 3 can be viewed as a form of semantic reduction [38,138] of the interferences allowed by the prior flow-insensitive approach in Algorithm 2. Specifically, the input environment to a load instruction in Algorithm 2 is the join of the set $S = \{\rho, \rho_1, \ldots, \rho_n\}$ where $\rho$ is the intra-thread environment and $\rho_1, \ldots, \rho_n$ are environments from interfering stores. The semantic-reduction operator we use in Algorithm 3 is to apply the transfer function of the load to each element of $S$ individually relative to all other loads (i.e., the Cartesian product). Therefore, the correctness of our algorithm directly follows the correctness argument in [38,138]. Additionally, we remove infeasible interferences combinations (Lines 31-33), which does not affect the soundness of the algorithm.

To see why our approach is a semantic-reduction, consider a load $l$ being executed with some set of interfering environments $I$ and intra-thread environment $\rho$. In Algorithm 2, the resulting environment $\rho'$ is calculated by:

$$\rho' = \text{TRANSFER}(l, \bigcup_{i \in I \cup \{\rho\}} i)$$

One semantic-reduction would be to apply the transfer function of the load on each environment individually.

$$\rho' = \bigcup_{i \in I \cup \{\rho\}} \text{TRANSFER}(l, i)$$
Our approach further delays the quantification over interferences. Specifically, during some iteration of Algorithm 3 (Lines 4–14), there exists a set $I_c$ of interference combinations. For a thread $g$, each iteration of Algorithm 3 calculates:

$$TE = \bigcup_{i_c \in I_c} \text{SEQABSINTMODIFIED-2}(g, i_c)$$

For these three equations, the input environment to a load is progressively subdivided\(^1\) into a finite number of subsets. Therefore, the our analysis is a semantic reduction and thus retains correctness from Algorithm 2 as shown in [121].

In the case of loops, the transfer function of a load can be executed more than once; each execution of the transfer function may use a different interference, so, using the same semantic-reduction operator would have resulted in a potentially infinite number of interference combinations. In this case, we conservatively merge all the *feasible* interferences into a single value. Correctness of this treatment directly follows the correctness of Algorithm 2.

In the case of aliasing, our algorithm can be lifted to use the output of any (sound) alias analysis by considering each alias-set as a single variable – it is a standard technique to handle aliasing in static analysis. In such case, our algorithm would operate on these alias-sets instead of on the individual program variables.

### 2.4 Checking Interference Feasibility with Constraints

We now present our new procedure for eliminating infeasible combinations of interferences. We revisit Algorithm 3 to show its integration with our new thread-modular analysis procedure.

Removing infeasible interferences from the thread-modular analysis significantly reduces computational overhead and increases accuracy. However, the main problem is that the feasibility checking has to be conducted efficiently for such an optimization to be useful. Therefore, our goal is to make the checking both *sound* and *efficient*. By sound, we mean that if the procedure determines a combination is infeasible then it is truly infeasible. By efficient, we mean that the procedure relies on constructing and solving a system of *lightweight* constraints, i.e., Horn clauses in finite domains, which can be decided using a Datalog engine in polynomial time.

Algorithm 4 shows the high-level flow of our feasibility analysis procedure. Initially, we traverse the set $Gs$ of control-flow graphs to compute a set $POs$ of constraints representing the order between statements which must hold on all possible executions of the program. We initialize the constraint system with these orderings by calling $\text{QUERY}\.\text{ADD}(POs)$.

During the execution of Algorithm 3 (Lines 31–33), for each $i_c \in I_c$, we compute a set $Cs$ of

\(^1\)Referred to as the *focusing* operator in Sagiv et al. [138].
Algorithm 4 Constraint-based feasibility checking.

1: \( \text{POs} \leftarrow \text{ProgramOrder-Constraints}(Gs) \)
2: \( \text{Query.Add(POs)} \)
3: function \( \text{Query.IsFeasible}(i_c) \) \( (\text{permutation of interferences}) \)
4: \( \text{Cs} \leftarrow \text{ReadsFrom-Constraints}(i_c) \)
5: \( \text{Query.Add(Cs)} \)
6: \( \text{res} \leftarrow \text{Query.Satisfiable()} \)
7: \( \text{Query.Remove(Cs)} \)
8: return \( \text{res} \)

reads-from constraints, which must be enforced in order to realize the interference combination \( i_c \). We add them to the system as well by calling \( \text{Query.Add(Cs)} \).

Our constraint analysis then, using a set of deduction rules, expands upon these input constraints to generate more constraints. We invoke \( \text{Query.Satisfiable} \) to check if the constraint system is satisfiable. The deduction rules are designed such that, if the system is not satisfiable, then \( i_c \) is guaranteed to be infeasible. In the remainder of this section, we go into each of these steps in detail.

2.4.1 The Program-Order and the Reads-From Constraints

To check the simultaneous feasibility of \( \text{POs} \) and \( \text{Cs} \), we first compute the dominators on a thread’s CFG. Given two nodes \( m \) and \( n \) in a graph \( g \), \( m \) dominates \( n \) if all paths from the entry of \( g \) to \( n \) go through \( m \). Then, we define the following relations:

- **Dominates** is the dominance relation on a thread’s CFG: \( (m, n) \in \text{Dominates} \) means \( m \) dominates \( n \).
- **NotReachable** is reachability on a thread’s CFG: \( (m, n) \in \text{NotReachable} \) means node \( m \) can not reach node \( n \).
- **ThCreates** is a parent–child relation: \( (p, n_{sta}) \in \text{ThCreates} \) if \( p \) is thread creation point and \( n_{sta} \) is the child thread’s start node.
- **ThJoins** is another parent–child relation: \( (p, n_{end}) \in \text{ThJoins} \) means \( p \) is a thread join on a child thread with node \( n_{end} \) as exit.
- \( (l, v) \in \text{IsLoad} \) means \( l \) is a load of variable \( v \).
- \( (s, v) \in \text{IsStore} \) means \( s \) is a store to variable \( v \).
- **ReadsFrom** is obtained from the combination \( i_c \) under test: \( (l, s) \in \text{ReadsFrom} \) if the load \( l \) is reading from the store \( s \).

All these relations can be computed from the given set \( Gs \) of control-flow graphs efficiently [111]. Furthermore, they are defined over finite domains (nodes or variables), which means constraints built upon these relations are efficiently decidable.
Figure 2.4: Deduction rules used by our feasibility analysis.

### 2.4.2 Using Deduction to Check Interference Feasibility

Figure 2.4 shows the deduction rules underlying our feasibility analysis. If a contradiction is reached after applying the rules to the input constraints, the interference combination is guaranteed to be infeasible. For brevity, we only present the intuition behind these rules. Detailed proofs can be found in our supplementary material.

Rules 2.1, 2.2, and 2.3 create the must-happen-before relation, MHB, where \((m,n) \in \text{MHB}\) means node \(m\) must-happen-before node \(n\) in the context of the current interference combination. Rule 2.4 is simply the transitive property for the must-happen-before relation.

First, if \(m\) dominates \(n\) in a CFG, since \(m\) occurs before \(n\) on all program paths, \(m\) must happen before \(n\) (Rule 2.1). We check if \(n\) can reach \(m\) to ensure that even if \(m\) dominates \(n\), \(m\) can never subsequently occur after \(n\) (e.g., if \(n\) is in a loop). Similarly, since a thread cannot execute before it is created, or after it terminates, ThCreates and ThJoins also map directly to MHB (Rule 2.2).

Rule 2.3 captures the scenario of two stores overwriting each other as shown in Figure 2.5. Here, one thread has stores \(s_1\) and \(s_2\), and a second thread has one load \(l\). ReadsFrom\((l, s_1)\) is represented by the dashed edge (flow of data) from \(s_1\) to \(l\). MHB\((s_1, s_2)\) is represented by the solid edge from \(s_1\) to \(s_2\). Given the two previous relations, the rule deduces the relation MHB\((l, s_2)\), represented by the red dotted edge. The implication is that for load \(l\) to read from the first store \(s_1\), \(l\) must happen before the second store \(s_2\).

The intuition behind this rule is that if \(s_2\) executes before \(l\), then \(s_2\) would overwrite the value
of \( s_1 \), making it impossible for \( l \) to read the value of \( s_1 \). Note that this must-happen-before constraint is only considered for \( i_c \), the current combination of interferences: it does not hold globally across all executions of the program.

Rule 2.5 introduces the \texttt{NotReadsFrom} relation. For a load store pair \((l, s) \in \texttt{NotReadsFrom}\) if in the current interference combination \( l \) cannot read from \( s \).

Rule 2.6 prevents a thread from reading an interference after it has been over-written; Figure 2.6 shows its application. The first thread has a store \( s_1 \), and the second thread has load \( l_1 \), store \( s_2 \), and then load \( l_2 \). Again, MHB relations are represented by solid edges, \texttt{ReadsFrom}(\( l_1, s_1 \)) is represented by the dashed edge, and the implied \texttt{NotReadsFrom}(\( l_2, s_1 \)) relation is represented by the red dotted edge.

Conceptually, the rule captures the situation when a value is read from an interference \((l_1: L1 = s)\), followed by a modification of the same memory location that was loaded \((s_2: s = L1 + 5)\), followed by a load of the same location \((l_2: L2 = s)\). Intuitively, since the interfering value was just overwritten, it cannot be loaded again. Therefore, the pair \((l_2, s_1)\) is added to \texttt{NotReadsFrom}.

Finally, our constraint analysis does not try to identify all infeasible combinations for efficiency reasons. However, the framework is generic enough to allow new rules and other types of constraint solvers to be plugged in easily to refine the approximation. We leave such extensions as future work.

### 2.4.3 The Running Example

We revisit the example in Figure 2.2 to illustrate our feasibility checking for one interference combination (Figure 2.7). Our goal is to decide if \texttt{ReadsFrom}(\( l_9, l_6 \)) and \texttt{ReadsFrom}(\( l_{11}, l_4 \)) can co-exist. At the start of the analysis, our constraint system would have the solid edges
Figure 2.7: Input and implied constraints for Figure 2.2.

from the MHB relations, which represent the program-order constraints, and the dashed edges from the READSFROM relations, which represent the current interference combination $i_c$.

First, we can deduce $\text{MHB}(l_{11}, l_5)$ by applying Rule 2.3: if $l_{11}$ does not happen before $l_5$, $l_5$ would overwrite the value of $x$, preventing $l_{11}$ from reading from $l_4$. This deduced MHB relation is represented by the red dotted edge in the figure.

Next, we can deduce a must-happen-before relation between $l_9$ and $l_6$ by applying Rule 2.4 twice. That is, $\text{MHB}(l_9, l_{11}) \land \text{MHB}(l_{11}, l_5)$ implies $\text{MHB}(l_9, l_5)$, followed by $\text{MHB}(l_9, l_5) \land \text{MHB}(l_5, l_6)$ implies $\text{MHB}(l_9, l_6)$. The result is represented by the red dotted edge from $l_9$ to $l_6$.

At this point, we have a contradiction: since $b1 = \text{flag}$ must-happen-before $\text{flag} = \text{true}$, $b1$ cannot read the value of $\text{true}$ (Rule 2.5). So, this interference combination is proved to be infeasible. (There are more implied edges in Figure 2.7; for clarity, we show only those relevant to the check.)

2.5 Clustering and Pruning Optimizations

To reduce the number of interference combinations, we apply dependency-based clustering analysis and property-directed pruning. Consider the program in Figure 2.8: the main thread creates two children in the function \text{thr} with arguments 5 and 10, respectively. The \text{thr} function performs a store to $x$ (Line 5) based on the value passed as an argument ($v$). At the load of $x$ in the \text{thr} function, the value may come from the initial value 0, from the main thread (Line 12), or from the other thread \text{thr} (Line 5). This results in three combinations of loads in \text{thr} to be tested on every iteration.

However, the reachability of \text{ERROR!} does not depend on the value loaded from $x$, since the error condition ($t1 < 0$) only depends on the argument passed to \text{thr}. As such, the load of $x$ is immaterial to the property. We can formally capture this notion of immateriality using control and data dependencies [51].

Intuitively, a statement $s$ is data dependent on $t$ if the value of $t$ may affect the computation of $s$. For example, in Figure 2.8, the statement $t1 = 5 \ast v$ is data dependent on the input
parameter v. On the other hand, a statement l is control dependent on m if the execution of m affects the reachability of l. For example, the ERROR! statement in Figure 2.8 is control-dependent on the evaluation of the predicate t1 < 0.

The composition of the control- and data-dependency relations is the program dependence graph [51]. Note that in concurrent programs, the dependency graph may span across multiple threads, due to the flow of data from shared memory writes to reads.

Next, we show two applications of the program dependence graph for optimizing our overall algorithm.

2.5.1 Property-Directing the Analysis

First, we create the backward slice on every property in the program. The backward slice of a program with respect to a property s is the set of nodes backward reachable from s in the program dependence graph; it contains all the statements involved in the computation of s (Theorem 2.2 [96]).

As an example, the program dependence graph for Figure 2.8 is shown in Figure 2.9. Dashed edges are control dependencies and solid edges are data dependencies. The backward slice on the ERROR! statement is also shown: the dotted nodes are nodes not contained in the slice. All computations involving x can be ignored, since the slice shows that they are irrelevant to the property being verified.

During our analysis, a statement not on the backward slice uses the identity function as its transfer function. Similarly, any load not on the backward slice can be ignored when computing interference combinations.

2.5.2 Clustering Interferences via Dependencies

Second, during the generation of combinations of interferences, we do not always consider the Cartesian product across all sets of loads. Instead, we group loads together to form cluster
Consider the program in Figure 2.10. Initially, \( x \) and \( y \) are zero; the first thread sets them to one, and the second thread checks the property that they are both greater than or equal to zero. The backward slice on assert\((x \geq 0)\) contains lines 4, 8, and 10. The backward slice on assert\((y \geq 0)\) contains lines 5, 9, and 11. Without optimization, in Algorithm 3, the loads on \( x \) and \( y \) both have two potential environments to read from: the interfering store and the environment within the thread. In total, there are \( 2 \times 2 = 4 \) combinations leading to four abstract interpreter executions.

The backward slices on properties in the program form disjoint subgraphs; e.g., a graph with the operations on \( x \) and those on \( y \). The interference combinations in the subgraphs can be considered independently requiring only \( \max(2, 2) = 2 \) interpreter executions.

### 2.6 Experiments

We implemented our flow-sensitive thread-modular analysis in a software tool named Watts, designed for verifying multithreaded programs in the LLVM intermediate language. All
experiments were performed on C programs written using POSIX threads. We used the Apron library [87] for implementing the sequential analyzer over interval and octagon abstract domains, and the Datalog solver in Z3 ($\mu$Z [79]) for solving the causality constraints.

We have evaluated WATTS on two sets of benchmark programs. The first set consists of multithreaded programs from SVCOMP [152]. The second set consists of various Linux device drivers from [154] and [47]. In all benchmark programs, the reachability properties are expressed in the form of embedded assertions. Table 2.2 shows the characteristics of these programs, including the name, the number of lines of code (LoC), the number of threads, and the number of assertions. In total, our benchmarks have 26,309 lines of code and 10,078 assertions. For the device driver benchmarks, in particular, since assertions are not included in the original source code, we manually added these assertions. We performed all experiments on a computer with 8 GB RAM and a 2.60 GHz CPU.

Although we used the device driver benchmarks from [47], the verification problem targeted by our method is significantly different from theirs. DUET assumes each device driver is a parametric program, where a thread routine may be executed by an unbounded number of threads. In contrast, our method is designed to analyze programs with a finite number of threads. As shown in Section 2.1, our method, using a set of control-flow graphs as opposed to a monolithic data-flow graph, is often more accurate than theirs. Therefore, during experiments, we do not directly compare WATTS with DUET. Instead, we focus on comparing our method with the state-of-the-art thread-modular approaches [52,119–121]. For evaluation purposes, we implemented both methods in WATTS: the flow-insensitive analysis of Algorithm 3 and the flow-insensitive analysis of Algorithm 2.

Table 2.3 shows the results of comparing Algorithm 3 and Algorithm 2 in the interval abstract domain. Column 1 shows the name of the benchmark. Columns 2–3 show the result of running Algorithm 2. Columns 4–5 show the result of running Algorithm 3 without using the feasibility checking. Columns 6–7 show the result with the feasibility checking. Columns 8–9 show the result with clustering/pruning optimizations. For each test case, Tm. is the run time in seconds and Verif. is the number of verified properties. The last row shows the sum of all columns.

Compared to the prior flow-insensitive approach (Columns 2–3), our baseline flow-sensitive method (Columns 4–5) can already achieve a 12x increase in the number of verified properties (from 38 to 452) without employing the lightweight constraint-based feasibility checking. This demonstrates the benefits of delaying the join operation across threads. Furthermore, the significant increase in accuracy comes at the modest 1.5x increase in run time.

With the constraint-based feasibility checking, a more significant improvement can be observed (Columns 6–7): there is a 28x increase in the number of verified properties (from 38 to 1,078) compared to the prior flow-insensitive approach. Furthermore, the large increase in accuracy comes with only an 1.6x increase in run time.

Finally, with the optimizations from Section 2.5, our method improves further (Columns 8–9).
Compared to the prior flow-insensitive approach (Columns 2–3), our method only has a 1.4x increase in the runtime overhead but with a 28x increase in number of verified properties. Compared to the version of our method without optimizations (Columns 6–7), the version with optimization finishes the entire analysis 1.4x faster. Additionally, the optimized version finishes slightly faster than the non-constraint based approach (Columns 4–5) while able to verify 2.4x as many properties.

Note that across all experiments, the number of verified properties are strictly increasing: e.g., the flow-sensitive approach with optimizations verifies all the properties of the flow-insensitive approach and more. At most we were able to verify 1,078 properties. Those we missed largely were due to cross-thread synchronization which was not captured by our constraint analysis.

In addition to the results in Table 2.3, we also performed experiments using the octagon abstract domain. We observed little increase in accuracy as a result of this change, indicating that the properties being verified are mostly on inter-thread concurrency control behavior, and therefore a more sophisticated representation of numerical relations over the program variables does not offer more advantages. For brevity, we omit the result table for the octagon domain.

In the past, efforts on introducing flow-sensitivity to static analysis procedures often result in scalability issues (e.g., [86]); however, this is not the case for our new method. Figure 2.11 shows our experiments on a parametrized program, where the run time of our new method grows only moderately with the increase in the number of threads. Here, the $x$-axis is the number of threads and the $y$-axis is the run time for all methods. The optimized constraint method has slightly lower runtime than the least accurate flow-insensitive approach. Furthermore, our new method also enjoys an almost linear growth in the execution time, indicating it is scalable.
2.7 Related Work

There is a large body of work on the static analysis and formal verification of multithreaded programs, but none of these existing methods can obtain flow sensitivity in thread-modular analysis with a reasonable run-time cost. For brevity, we discuss only those that are most relevant to our new method. The interested reader can see Rinard [136] for a survey of early work.

Thread-modular abstract interpretation was introduced by Ferrara [52] and Miné [119,120]. As shown, their approaches eagerly joined interferences and considered them flow-insensitively, thus introducing inaccuracies. Our method avoids such drawbacks.

Ferrara [52] also introduced models designed specific for the Java memory model to remove certain types of infeasible interferences in an ad hoc fashion. Our constraint-based feasibility checking, in contrast, is more general and systematic, and can handle transitive must-happen-before constraints as well as other constraints both within and across threads.

Miné [121] introduced an extension to their prior thread-modular analysis to compute relational interferences. This allows for relations between variables to be maintained across threads, thereby bringing more accuracy than using non-relational interferences. However, as we have explained earlier, this technique is orthogonal and complementary to our new method.

Farzan and Kincaid [47] introduced a method to iteratively construct a monolithic data-flow graph for a concurrent program. However, their technique, as well as similar methods designed for parametric programs [93,104], targets the problem of verifying properties in a concurrent program with an unbounded number of threads. As we have shown earlier, our new method is often significantly more accurate than these existing methods.

Thread-modular analyses have also been applied in the context of both model checking [59,77] and symbolic execution [146,147]. However, these approaches are, in general, are either heavyweight or under-approximative, and therefore are complementary to our abstract-interpretation based approach.

2.8 Discussion

This chapter presented the first constraint-based flow-sensitive method for composing standard abstract interpreters to form a more accurate thread-modular analysis for concurrent programs. Our method relies on constructing and solving a system of lightweight happens-before constraints to decide the feasibility of inter-thread interference combinations. We also use clustering and pruning to reduce the run-time overhead of our thread-modular analysis. We have implemented our new method in a software tool called Watts and evaluated it on a set of multithreaded C programs. Our experimental results show that the new method can
significantly increase the accuracy of the thread-modular analysis while maintaining a modest run-time overhead.
Table 2.2: Statistics of the benchmarks in our experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>Threads</th>
<th>Properties</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread01</td>
<td>29</td>
<td>3</td>
<td>1</td>
<td>created</td>
</tr>
<tr>
<td>create01</td>
<td>24</td>
<td>2</td>
<td>1</td>
<td>created</td>
</tr>
<tr>
<td>create02</td>
<td>28</td>
<td>2</td>
<td>1</td>
<td>created</td>
</tr>
<tr>
<td>sync01</td>
<td>38</td>
<td>3</td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>sync02</td>
<td>36</td>
<td>3</td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>intra01</td>
<td>41</td>
<td>3</td>
<td>1</td>
<td>created</td>
</tr>
<tr>
<td>dekker1</td>
<td>65</td>
<td>3</td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>fk2012</td>
<td>88</td>
<td>3</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>keybISR</td>
<td>62</td>
<td>3</td>
<td>2</td>
<td>154</td>
</tr>
<tr>
<td>ib700_01</td>
<td>346</td>
<td>3</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>ib700_02</td>
<td>466</td>
<td>23</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>ib700_03</td>
<td>587</td>
<td>41</td>
<td>81</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>i8xxtco_01</td>
<td>735</td>
<td>3</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>i8xxtco_02</td>
<td>901</td>
<td>22</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>i8xxtco_03</td>
<td>1027</td>
<td>42</td>
<td>103</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>machz_01</td>
<td>667</td>
<td>8</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>machz_02</td>
<td>795</td>
<td>29</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>machz_03</td>
<td>881</td>
<td>41</td>
<td>83</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>mix_01</td>
<td>457</td>
<td>12</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>mix_02</td>
<td>580</td>
<td>31</td>
<td>62</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>pcwd_01</td>
<td>1197</td>
<td>8</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>pcwd_02</td>
<td>1405</td>
<td>41</td>
<td>81</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>sbc_01</td>
<td>686</td>
<td>24</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>sc1200_01</td>
<td>715</td>
<td>24</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>sc1200_02</td>
<td>765</td>
<td>31</td>
<td>93</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>smsc_01</td>
<td>904</td>
<td>12</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>smsc_02</td>
<td>931</td>
<td>12</td>
<td>24</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>sc520_01</td>
<td>806</td>
<td>4</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>sc520_02</td>
<td>880</td>
<td>41</td>
<td>81</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wfwdt_01</td>
<td>777</td>
<td>4</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wfwdt_02</td>
<td>907</td>
<td>51</td>
<td>101</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wdt</td>
<td>1023</td>
<td>31</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wdt977_01</td>
<td>867</td>
<td>16</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wdt977_02</td>
<td>877</td>
<td>31</td>
<td>92</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wdt_pci</td>
<td>1133</td>
<td>31</td>
<td>1</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>wdt_pci02</td>
<td>1165</td>
<td>31</td>
<td>122</td>
<td>[47], added asserts</td>
</tr>
<tr>
<td>pcwdpci_01</td>
<td>1363</td>
<td>64</td>
<td>128</td>
<td>[47], added asserts</td>
</tr>
</tbody>
</table>
Table 2.3: Experimental results in the interval domain.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tm. (s) Verif.</td>
<td>Tm. (s) Verif.</td>
<td>Tm. (s) Verif.</td>
<td>Tm. (s) Verif.</td>
</tr>
<tr>
<td>thread01</td>
<td>0.03 0</td>
<td>0.05 0</td>
<td>0.05 1</td>
<td>0.09 1</td>
</tr>
<tr>
<td>create01</td>
<td>0.02 0</td>
<td>0.03 0</td>
<td>0.04 1</td>
<td>0.07 1</td>
</tr>
<tr>
<td>create02</td>
<td>0.03 0</td>
<td>0.03 0</td>
<td>0.03 1</td>
<td>0.07 1</td>
</tr>
<tr>
<td>sync01</td>
<td>0.04 0</td>
<td>0.05 1</td>
<td>0.06 1</td>
<td>0.07 1</td>
</tr>
<tr>
<td>sync02</td>
<td>0.04 0</td>
<td>0.06 0</td>
<td>0.07 1</td>
<td>0.07 1</td>
</tr>
<tr>
<td>intra01</td>
<td>0.03 0</td>
<td>0.03 0</td>
<td>0.03 1</td>
<td>0.08 1</td>
</tr>
<tr>
<td>dekker1</td>
<td>0.14 0</td>
<td>9.81 0</td>
<td>2.10 1</td>
<td>0.75 1</td>
</tr>
<tr>
<td>fk2012</td>
<td>0.10 0</td>
<td>0.25 0</td>
<td>0.25 1</td>
<td>0.18 1</td>
</tr>
<tr>
<td>keybISR</td>
<td>0.05 0</td>
<td>0.15 0</td>
<td>0.14 2</td>
<td>0.12 2</td>
</tr>
<tr>
<td>ib700_01</td>
<td>0.09 0</td>
<td>0.10 0</td>
<td>0.10 1</td>
<td>0.13 1</td>
</tr>
<tr>
<td>ib700_02</td>
<td>1.17 0</td>
<td>0.88 0</td>
<td>0.95 1</td>
<td>1.03 1</td>
</tr>
<tr>
<td>ib700_03</td>
<td>33.46 0</td>
<td>40.95 40</td>
<td>36.95 81</td>
<td>37.81 81</td>
</tr>
<tr>
<td>i8xxtco_01</td>
<td>0.15 0</td>
<td>0.13 0</td>
<td>0.13 1</td>
<td>0.22 1</td>
</tr>
<tr>
<td>i8xxtco_02</td>
<td>1.34 0</td>
<td>0.96 0</td>
<td>1.02 1</td>
<td>1.24 1</td>
</tr>
<tr>
<td>i8xxtco_03</td>
<td>38.07 18</td>
<td>50.78 61</td>
<td>47.90 103</td>
<td>55.24 103</td>
</tr>
<tr>
<td>machz_01</td>
<td>0.21 0</td>
<td>0.18 0</td>
<td>0.18 1</td>
<td>0.29 1</td>
</tr>
<tr>
<td>machz_02</td>
<td>0.97 0</td>
<td>0.69 0</td>
<td>0.76 1</td>
<td>0.94 1</td>
</tr>
<tr>
<td>machz_03</td>
<td>41.30 0</td>
<td>74.32 42</td>
<td>153.50 83</td>
<td>118.25 83</td>
</tr>
<tr>
<td>mix_01</td>
<td>0.24 0</td>
<td>0.19 0</td>
<td>0.20 1</td>
<td>0.29 1</td>
</tr>
<tr>
<td>mix_02</td>
<td>12.42 1</td>
<td>15.22 31</td>
<td>13.24 62</td>
<td>15.28 62</td>
</tr>
<tr>
<td>pcwd_01</td>
<td>0.25 0</td>
<td>0.21 0</td>
<td>0.21 1</td>
<td>0.32 1</td>
</tr>
<tr>
<td>pcwd_02</td>
<td>33.12 0</td>
<td>41.57 40</td>
<td>33.77 81</td>
<td>38.23 81</td>
</tr>
<tr>
<td>sbc_01</td>
<td>0.60 0</td>
<td>0.73 0</td>
<td>1.09 1</td>
<td>0.57 1</td>
</tr>
<tr>
<td>sc1200_01</td>
<td>0.53 0</td>
<td>0.62 0</td>
<td>0.47 1</td>
<td>0.54 1</td>
</tr>
<tr>
<td>sc1200_02</td>
<td>70.46 0</td>
<td>119.24 62</td>
<td>161.00 93</td>
<td>122.48 93</td>
</tr>
<tr>
<td>smsc_01</td>
<td>0.35 0</td>
<td>0.32 0</td>
<td>0.40 1</td>
<td>0.51 1</td>
</tr>
<tr>
<td>smsc_02</td>
<td>3.73 0</td>
<td>7.27 1</td>
<td>15.39 24</td>
<td>6.12 24</td>
</tr>
<tr>
<td>sc520_01</td>
<td>0.64 0</td>
<td>1.23 0</td>
<td>0.73 1</td>
<td>0.72 1</td>
</tr>
<tr>
<td>sc520_02</td>
<td>50.87 0</td>
<td>81.27 39</td>
<td>65.95 81</td>
<td>46.95 81</td>
</tr>
<tr>
<td>wfwdt_01</td>
<td>0.61 0</td>
<td>1.20 0</td>
<td>0.71 1</td>
<td>0.70 1</td>
</tr>
<tr>
<td>wfwdt_02</td>
<td>94.54 0</td>
<td>148.32 0</td>
<td>118.39 101</td>
<td>83.91 101</td>
</tr>
<tr>
<td>wdt</td>
<td>0.71 0</td>
<td>0.49 0</td>
<td>0.55 1</td>
<td>0.69 1</td>
</tr>
<tr>
<td>wdt977_01</td>
<td>0.57 0</td>
<td>0.44 0</td>
<td>0.49 1</td>
<td>0.65 1</td>
</tr>
<tr>
<td>wdt977_02</td>
<td>51.86 0</td>
<td>58.92 32</td>
<td>86.16 93</td>
<td>92.01 93</td>
</tr>
<tr>
<td>wdt_pci</td>
<td>0.79 0</td>
<td>0.55 0</td>
<td>0.61 1</td>
<td>0.77 1</td>
</tr>
<tr>
<td>wdt_pci02</td>
<td>75.14 1</td>
<td>114.55 31</td>
<td>100.12 122</td>
<td>110.33 122</td>
</tr>
<tr>
<td>pcwdpci_01</td>
<td>91.10 18</td>
<td>115.82 72</td>
<td>136.13 128</td>
<td>109.02 128</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>605.77</strong> 38</td>
<td><strong>887.61</strong> 452</td>
<td><strong>979.87</strong> 1,078</td>
<td><strong>846.74</strong> 1,078</td>
</tr>
</tbody>
</table>
Chapter 3

Thread-Modular Abstract Interpretation for Weak Memory

Concurrent software written for modern computer architectures, though ubiquitous, remains challenging for static program-analysis. Although abstract interpretation [37] is a powerful static analysis technique and prior thread-modular methods [52, 101, 119–121] mitigated interleaving explosion, none were 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.

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 [11]. 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 3.1 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 two 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 not only the reordering of a write to variable x with a subsequent read from variable 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...
void thread1() {
    x = 1;
    a = y;
}

assert(a != 0 && b != 0);

void thread2() {
    y = 1;
    b = x;
}

Figure 3.1: The assertion holds under SC, but not under TSO, PSO, and RMO memory models.

in Section 3.1.

Broadly speaking, existing thread-modular abstract interpreters fall into two categories, neither modeling weak-memory related behaviors. The first are SC-specific [47,48,101]: they are designed to be flow-sensitive in terms of modeling thread interactions but consider only behaviors compatible with the SC memory. The second [119–121] are oblivious to memory models (MM-oblivious): they permit all orderings of memory-writes across threads. Therefore, MM-oblivious methods may report many spurious errors (bogus alarms) whereas SC-specific methods, although more accurate for SC memory, may miss many real errors on weaker memory (bogus proofs). This flaw is not easy to fix using conventional approaches [121]. For example, maintaining relational invariants at all program points makes the analysis prohibitively expensive. In Section 3.1, we use examples to illustrate issues related to these existing 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 3.2. In this figure, the boxes with bold text highlight our main contributions.

Our method is advantageous since it builds on a new unified framework modeling 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 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 [39,47,100,117], or not specifically targeting weak memory [101,119–121].

Our method also differs significantly from techniques designed for bug hunting as opposed to obtaining correctness proofs. For example, in systematic concurrency testing, stateless model checking [58,66] was extended from SC to weaker memory models [6,9,41,82,129,171]. In bounded model checking, Alglave et al. [16] modeled weak memory through code
transformation or direct symbolic encoding [15,17]. 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 [12] as the C front-end, Apron [87] for abstract domains, and the µZ [79] Datalog engine in Z3 [40]. We evaluated FruitTree on a total of 209 litmus tests, and 52 larger multithreaded programs with a total of 61,981 lines of C code, where reachability properties are expressed as assertions embedded in the program code. 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.

We evaluated FruitTree against the MM-oblivious analyzer of Miné [121], the SC-specific analyzer Watts [101], and a non thread-modular analyzer Duet [47,48]. 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 the first memory-model aware thread-modular abstract interpreter.
- We introduce a declarative analysis deducing the feasibility of thread-interferences under a variety of memory models.
- We implement our method and evaluate it on a set of benchmarks showing its high accuracy and moderate overhead.

The remainder of this paper is organized as follows. First, we motivate our technique via examples in Section 3.1. Then, we provide background on memory models, and abstract interpretation in Section 3.2. We present our new declarative analysis checking the feasibility
of thread inferences in Section 3.3, followed by the main algorithm for thread-modular abstract interpretation in Section 3.4. We present our experimental results in Section 3.5, review related work in Section 3.6, and conclude in Section 3.7.

### 3.1 Motivating Examples

Consider the program in Figure 3.3. 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 generate bogus errors, why SC-specific ones generate bogus proofs, and how our new method fixes both issues.

#### 3.1.1 The Example Under SC, TSO, PSO, and RMO

First, note that the assertion in Figure 3.3 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 [11]. TSO permits the delay of a store after a subsequent load to a disjoint memory address (as in Figure 3.1). 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 3.3). 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. Thus, the second thread may read 10 from $y$ before 5 is written to $x$ in global memory causing the assertion to fail.

The fence 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
Figure 3.4: Comparing the effectiveness of methods in handling the example programs in Figure 3.1 and Figure 3.3.

holds again.

### 3.1.2 Where Prior Techniques are Ineffective

MM-oblivious methods [119–121] 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 works 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, the value may come from any one of these thread interferences. This iterative process repeats until a fixed point is reached.

Next, we demonstrate an MM-oblivious analyzer on Figure 3.3. Consider the thread interferences to be a map from variables to abstract values in the interval domain [37]. 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 the 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 [47, 48, 101] 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 3.3 under PSO or RMO when the fence is removed.

Figure 3.4 summarizes the ineffectiveness of prior techniques on the programs in Figures 3.1
and 3.3 with and without fences. Note that in Figure 3.1 the fence instruction may be added between the write and read instructions of both threads. This shows how prior MM-oblivious methods report bogus alarms, prior SC-specific methods report bogus proofs, while our new method eliminates both.

### 3.1.3 A New Way to Consider Weak Memory Models

Typically, 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. Consider Figure 3.3: 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. First is the flow-sensitive propagation of thread interferences. Instead of eagerly joining all interfering stores, we handle each combinations separately. The second is a declarative modeling of consistency semantics general enough to capture SC, TSO, PSO, and RMO [11, 148, 161]. Together, these prune infeasible combinations such as \( x \) and \( y \) reading 0 and 10, respectively, in Figure 3.3.

Our new method analyzes thread 2 in Figure 3.3 by considering four different interference combinations, \( \rho_1 - \rho_4 \), separately.

- \( \rho_1 = y \mapsto [0, 0] \) and \( x \mapsto [0, 0] \),
- \( \rho_2 = y \mapsto [10, 10] \) and \( x \mapsto [5, 5] \),
- \( \rho_3 = y \mapsto [0, 0] \) and \( x \mapsto [5, 5] \),
- \( \rho_4 = y \mapsto [10, 10] \) 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. \( \rho_1, \rho_2, \) and \( \rho_3 \) 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 3.3. For now, consider \( \rho_4 \) in Figure 3.3: 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 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 is a proof that \( \rho_4 \) can not exist in any concrete execution. Since the
combinations $\rho_{1-3}$ do not violate the assertion, and $\rho_4$ is proved to be infeasible, the program is verified correct.

## 3.2 Background

Here, we review weak memory models and abstract interpretation.

### 3.2.1 Modeling Concurrency

A concurrent program consists of a finite set of threads. Each thread accesses a set $\{a, b, c, \ldots\}$ of local variables. All threads access a set $\{x, y, z, \ldots\}$ of global variables. Threads use load and store instructions to read/write global variables. A thread creates a child thread with ThreadCreate, and waits it to terminate with ThreadJoin.

We represent a program using a set $G = \{G_1, \ldots, G_k\}$ of flow graphs. Each graph $G \in G$, where $G = (N, n_0, \delta)$, 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 $\delta$ is the transition relation. That is, $(n', n) \in \delta$ iff there exists an edge from $n'$ to $n$.

We assume each $n \in \mathbb{N}$ is associated with an atomic instruction which 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. When accessing variables on the global memory, threads may use a special fence instruction to impose strict program order between memory operations issued before and after the fence.

### 3.2.2 Consistency of Memory Models

The simplest memory model is sequential consistency (SC) [107]. 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 case: memory models are often weaker than SC, and can be characterized by their relaxations of the program-order and write-atomicity requirements as shown in Figure 3.5. Here, $R(v_1) \rightarrow W(v_2)$ is a read of $v_1$ followed by a write of $v_2$ within a thread.
We use $S$ to denote the set of all memory environments. $S$ is a lattice with properly defined top ($\top$) and bottom ($\bot$) elements, join ($\sqcup$), partial-order ($\sqsubseteq$), and a widening operator [37]. Each node $n \in N$ has a transfer function $t : S \to S$, taking an environment $s' \in S$ as input (before executing the atomic operation in $n$) and returns a new environment $s \in S$ as output.

\footnote{For ease of presentation we assume a variable maps to a single value. Our analysis can trivially use relational domains.}
Let $tfunc \in \mathbb{N} \rightarrow (\mathbb{S} \rightarrow \mathbb{S})$ 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 $s = (tfunc(n))(s') = \{x \mapsto [3, 6], a \mapsto [2, 5]\}$.

The goal of an abstract interpreter is to compute an environment map $M \in (\mathbb{N} \rightarrow \mathbb{S})$ 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 $tfunc(n)$ and joins the resulting environments for all $n$, until they reach a fixed point.

Without getting into the details (refer to the literature [37]), we define the sequential analyzer as a fixed-point computation with respect to the function $AnalyzeSeq : (\mathbb{N} \rightarrow \mathbb{S}) \rightarrow (\mathbb{N} \rightarrow \mathbb{S})$:

$$AnalyzeSeq(M) = n \mapsto (tfunc(n))\left( \bigsqcup_{(n', n) \in \delta} M(n') \right)$$

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. $(tfunc(n))(s')$ 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.

Directly applying the sequential analyzer to a multithreaded program is not practical because it leads to an exponential complexity wrt the number of threads. Instead, thread-modular techniques [101,119–121] iteratively apply $AnalyzeSeq$ to each thread, as if they were sequential programs, 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 $T$, the interferences are the effects of global memory stores from all $t' \in T$. The interferences are a map $(\mathbb{V} \rightarrow 2^\mathbb{S})$ from each variable $v \in \mathbb{V}$ read by thread $t$ to the set of memory environments 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 [119–121] 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. In the remainder of this paper, we present our method for mitigating this problem.
3.3 Constraint-Based Interference-Feasibility Checking for Relaxed-Memory Models

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 \( \text{ic} = \{(l, s), \ldots\} \) where each \((l, s) \in \text{ic}\) is a load \(l\) and an interfering store \(s\).

Checking the feasibility of \(\text{ic}\) 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 set \(\{(l, s) | (l, s) \in \text{ReadsFrom}\}\), and outputs the relation \(\text{MustNotReadFrom}\). \((l, s) \in \text{MustNotReadFrom}\) 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 \(\text{ReadsFrom}\) relation in \(\text{ic}\).

Consider the program in Figure 3.3 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 \(l_y, s_y^{10}, l_x, s_x^0\), respectively. Then, the feasibility problem is stated as follows: given \((l_y, s_y^{10}) \in \text{ReadsFrom}\), check if \((l_x, s_x^0) \in \text{MustNotReadFrom}\).

### 3.3.1 Preliminaries

Before presenting the details of our feasibility checking procedure, we define a set of unary and binary relations over instructions and program variables. Specifically, \((s_1, v_1) \in \text{IsLoad}\) denotes \(s_1\) is a load of variable \(v_1\), and \((s_2, v_2) \in \text{IsStore}\) denotes \(s_2\) is a store to variable \(v_2\). We use \((s_1, \_ \_ ) \in \text{IsLoad}\) if we do not care about the variable. Similarly, we use \(f \in \text{IsFence}\) to denote that \(f\) is a fence. We also use \(\text{IsLLMembar}, \text{IsLSMembar}, \text{IsSLMembar}, \text{IsSSMembar}\) to denote load–load, load–store, store–load, and store–store memory barriers as defined in the SPARC architecture [161]; for example, a load–store \text{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 (\(\text{Dominates}, \text{NotReachableFrom}, \text{ThreadCreates}, \text{ThreadJoins}\)) 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 \(\text{ReadsFrom}\) relation comes from the given \(\text{ic}\), from which we deduce the \(\text{MustNotReadFrom}\) relation.
Consider Figure 3.3 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 \((l_y, s_10_y) \in \text{ReadsFrom}\). Then, we deduce the \text{MustNotReadFrom} relation. Finally, we check if \((l_x, s_0_x) \in \text{MustNotReadFrom}\).

### 3.3.2 How the Program Order Constraint can be Relaxed

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

We define the rules for \( \text{NoReorder} \) given the allowed program-order relaxations for different memory models (Figure 3.5).

For SC, \( \text{NoReorder} \) is defined as:

\[
\frac{T}{(s_1, s_2) \in \text{NoReorder}} \quad \text{(under SC)}
\]

That is, no reordering is ever allowed under SC (row SC Figure 3.5).

For TSO, \( \text{NoReorder} \) is defined as:

\[
\frac{(s_1, \_)}{(s_1, s_2) \in \text{NoReorder}} \quad \text{(under TSO)}
\]

\[
\frac{(s_2, \_)}{(s_1, s_2) \in \text{NoReorder}} \quad \text{(under TSO)}
\]

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 3.5), while the second disallows Columns 3, 5, 7, and 9. Thus, reordering is permitted in two cases: Columns 4 and 8. \( W(v_1) \rightarrow R(v_1) \) (Column 4) may be reordered in our analysis under TSO (and PSO and RMO) for soundness: it permits \text{read-own-write-early} behaviors. We detail this shortly in Section 3.3.6.

For PSO, \( \text{NoReorder} \) is defined as:

\[
\frac{(s_1, \_)}{(s_1, s_2) \in \text{NoReorder}} \quad \text{(under PSO)}
\]
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 in the remaining three cases (Columns 4, 8, and 9).

For RMO, the inference rules are defined as:

\[
\begin{align*}
(s_1, v_1) &\in \text{IsLoad} \land (s_2, v_1) \in \text{IsLoad} & (s_1, s_2) \in \text{NoReorder} & (\text{under RMO})
\end{align*}
\]

\[
\begin{align*}
(s_1, v_1) &\in \text{IsLoad} \land (s_2, v_1) \in \text{IsStore} & (s_1, s_2) \in \text{NoReorder} & (\text{under RMO})
\end{align*}
\]

\[
\begin{align*}
(s_1, v_1) &\in \text{IsStore} \land (s_2, v_1) \in \text{IsStore} & (s_1, s_2) \in \text{NoReorder} & (\text{under RMO})
\end{align*}
\]

Similarly, the above inference rules can be directly translated from Columns 2, 3, and 6 of the table in Figure 3.5.

### 3.3.3 Modeling Memory Barriers and Fences

Next, we present the ordering constraints imposed by fences and memory barriers. We consider four variants of the `membar` instruction preventing loads and/or stores before the `membar` from being reordered with subsequent loads and/or stores [161].

\[
\begin{align*}
m \in \text{IsLLMembar} \land (s_1, \_ ) \in \text{IsLoad} \land (s_2, \_ ) \in \text{IsLoad} \\
\land (s_1, m) \in \text{Dominates} \land (m, s_2) \in \text{Dominates}
\end{align*}
\]

\[
\begin{align*}
(s_1, s_2) \in \text{NoReorder}
\end{align*}
\]

\[
\begin{align*}
m \in \text{IsLSMembar} \land (s_1, \_ ) \in \text{IsLoad} \land (s_2, \_ ) \in \text{IsStore} \\
\land (s_1, m) \in \text{Dominates} \land (m, s_2) \in \text{Dominates}
\end{align*}
\]

\[
\begin{align*}
(s_1, s_2) \in \text{NoReorder}
\end{align*}
\]

\[
\begin{align*}
m \in \text{IsSLMembar} \land (s_1, \_ ) \in \text{IsStore} \land (s_2, \_ ) \in \text{IsLoad} \\
\land (s_1, m) \in \text{Dominates} \land (m, s_2) \in \text{Dominates}
\end{align*}
\]

\[
\begin{align*}
(s_1, s_2) \in \text{NoReorder}
\end{align*}
\]

\[
\begin{align*}
m \in \text{IsSSMembar} \land (s_1, \_ ) \in \text{IsStore} \land (s_2, \_ ) \in \text{IsStore} \\
\land (s_1, m) \in \text{Dominates} \land (m, s_2) \in \text{Dominates}
\end{align*}
\]

\[
\begin{align*}
(s_1, s_2) \in \text{NoReorder}
\end{align*}
\]

We model fences in terms of `membars` since they prevent loads and stores from being reordered with subsequent loads and stores.

\[
\begin{align*}
f \in \text{IsFence} \\
f \in \text{IsLLMembar} \\
f \in \text{IsLSMembar}
\end{align*}
\]

\[
\begin{align*}
f \in \text{IsFence} \\
f \in \text{IsLSMembar}
\end{align*}
\]
\[
\begin{align*}
(s, s_{sta}) & \in \text{THREADCREATES} & (s, s_{sta}) & \in \text{MHB} \\
(s_{sta}) & \in \text{MHB} & (s, s_{end}) & \in \text{THREADJOINS} \\
(s_{end}, s) & \in \text{MHB} \\
(s_{1}, s_{2}) & \in \text{DOMINATES} \land (s_{2}, s_{1}) & \in \text{NOTREACHABLEFROM} \\
\land (s_{1}, s_{2}) & \in \text{NOREORDER} \\
(s_{1}, s_{2}) & \in \text{MHB} & (s_{1}, s_{3}) & \in \text{MHB} \\
\land (s_{1}, s_{3}) & \in \text{MHB} \\
(l, s_{1}) & \in \text{READSFROM} \land (s_{1}, s_{2}) & \in \text{MHB} \\
\land (l, v) & \in \text{ISLOAD} \land (s_{1}, v) & \in \text{ISSTORE} \land (s_{2}, v) & \in \text{ISSTORE} \\
(l, s_{2}) & \in \text{MHB} \\
\end{align*}
\]

Figure 3.6: Deduction rules for MHB (must-happen-before).

\[
\begin{align*}
(l, s) & \in \text{MHB} \\
(l, s) & \in \text{MUSTNOTREADFROM} \\
(l_{1}, s_{1}) & \in \text{READSFROM} \land (l_{1}, s_{2}) & \in \text{MHB} \land (s_{2}, l_{2}) & \in \text{MHB} \\
\land (l_{1}, v) & \in \text{ISLOAD} \land (l_{2}, v) & \in \text{ISLOAD} \land (s_{2}, v) & \in \text{ISSTORE} \\
(l_{2}, s_{1}) & \in \text{MUSTNOTREADFROM} \\
\end{align*}
\]

Figure 3.7: Deduction rules for the MUSTNOTREADFROM.

In addition, there are fences implicitly added to thread routines such as lock/unlock and signal/wait. For example, in the code snippet \(x = 1; \text{lock(lk); } a = y; \text{unlock(lk)},\) there is a fence inside lock(lk), ensuring \(x = 1\) always takes effect before \(a = y\). This is how most modern programming systems guarantee data-race-freedom [10] to application-level code (i.e., programs without data races have only SC behaviors). We model every call \(s_{c}\) to a POSIX thread routine using \(s_{c} \in \text{IsFence}\).

### 3.3.4 Deducing Interference Infeasibility

We divide our inference rules into two groups. The first (Figure 3.6) use the relations THREADCREATES, THREADJOINS, DOMINATES, and NOREORDER and generates the must-happen-before (MHB) relation.

Rule (3.1) 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 is \(s_{end}\), then \(s_{end}\) must happen before \(s\). The correctness of this rule is obvious.
Rule (3.2) 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 3.8 exemplifies this rule: the loop in the left CFG is outside the $\text{DOMINATES}$ edge, thus $(s_1, s_2) \in \text{NOTREACHABLEFROM}$. The loop in the right CFG encompasses the $\text{DOMINATES}$ edge, thus $(s_1, s_2) \notin \text{NOTREACHABLEFROM}$. The correctness of this rule is obvious.

Rule (3.3) 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 (3.4) 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 3.9 exemplifies this rule. Its correctness is obvious.

The second group of inference rules (Figure 3.7) takes the relations MHB and READS-FROM and generates the MUSTNOTREADFROM relation. Recall that if a load-store pair $(l, s) \in \text{MUSTNOTREADFROM}$, the value stored by $s$ can never flow to $l$. Thus, MUSTNOTREADFROM may be used to eliminate infeasible data flows.

Rule (3.5) 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 (3.6) 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 3.10 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 $s_1$. 
Figure 3.9: Example illustrating Rule (2.3)

Figure 3.10: Example illustrating Rule (3.6).
3.3.5 Proofs

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

In the remainder of this section, we explain why our deductive procedure is sound. First, the deduction of the \texttt{NoReorder} relation relaxing the program order requirement, from Figure 3.5, is sound.

**Theorem 1.** Let \(s_1\) and \(s_2\) be two instructions in the same thread. If our rules deduce \((s_1, s_2) \in \texttt{NoReorder}\), 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 \texttt{NoReorder} directly follow the memory model semantics provided by Adve and Gharachorloo [11] in Figure 3.5.

Next, we note that, given the \texttt{ReadsFrom} relation, the deduction of the \texttt{MustNotReadFrom} relation is sound.

**Theorem 2.** Let \(l\) and \(s\) be two instructions. If our rules deduce to \((l, s) \in \texttt{MustNotReadFrom}\), 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 argued why each rule is correct. More formal proofs can be obtained via contradiction. We omit the details for brevity.

3.3.6 Allowing Writes to be Non-Atomic

Our method soundly models \textit{buffer forwarding}, which corresponds to the write-atomicity requirement (Column 10 Figure 3.5). This allows a thread to read its own write before the written value is flushed to the memory, thus becoming visible to other threads. Buffer forwarding allows a thread to read the value from a previous store before it is flushed to memory. This is modeled in both the thread-level analyzer (\texttt{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 of NoREORDER (Section 3.3.2) always permits the reordering of a store with a subsequent load of the same variable (Column 4, Figure 3.5). That is, if \((s_1, v_1) \in \text{IsStore}\) and \((s_2, v_1) \in \text{IsLoad}\), we do not deduce \((s_1, s_2) \in \text{NoREORDER}\). Semantically, this is a consequence of buffer forwarding: within a thread \(t\), it may appear the store and load are reordered from the perspective of all threads \(t' \neq t\). Forbidding this reordering is equivalent to forcing a full flush of the store-buffers before every load, thus prohibiting any thread from reading its own store early.

```c
1  void thread1() {
2      x = 1; // s1
3      a = x; // s2
4      b = y; // s3
5  }
```

```c
1  void thread2() {
2      y = 1; // s4
3      fence; // s5
4      c = x; // s6
5  }
```

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

Figure 3.11: Write atomicity example under TSO.

Figure 3.11 exemplifies the requirement of this relaxation. First, the assertion may be violated under TSO. An error trace is: \(x = 1; a = 1; b = 0; y = 1; \text{flush } y; c = 0; \text{flush } x\). 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 in NoREORDER, preventing the assertion from unsound verification.

### 3.4 Analyzing Interferences Within Thread-Modular Abstract Interpretation

Next, we present the integration of the interference analysis (Section 3.3) with the thread-modular analyzer. The thread-modular analyzer is covered in full detail by Kusano and Wang [101]: our presentation will be terse. Our main contribution is a technique deducing infeasible interference combinations for weak-memory models: our method is sound for both SC and 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 \(n\) 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 \( \text{Interfs}' \).

\[
\text{Interfs}'(G, M, I) = l \mapsto \{e\} \cup I(l)
\]

where \( e \) is the environment after a remote store \( s \not\in n(G) \)
to the same variable as loaded by load \( l \in n(G) \)

\[
\text{Interfs}(G, M) = \text{lfp}(\text{Interfs}'(G, M), I_\perp)
\]

We use \( \text{Interfs}'(G, M, I) \) as shorthand for \( \text{Interfs}'(G, M)(I) \), where \( \text{Interfs}'(G, M) \) 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 \( \{\perp\} \). \( \text{lfp} \) computes the least-fixed point. \( \text{Interfs}' \) 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 3.3 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 \( \text{Interfs} \), 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 (\( \text{AnalyzeSeq} \), Section 2.2) to use interference-combinations. \( \text{AnalyzeTM}' \) 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 \( M(n) \).

\[
\text{AnalyzeTM}'(G, ic, M) = n \mapsto \text{tfunc}(n)(e)
\]

where \( e = \bigcup_{(n', n) \in t(G)} M(n') \sqcup ic(n) \)

\[
\text{AnalyzeTM}(G, ic) = \text{lfp}(\text{AnalyzeTM}'(G, ic), M_\perp)
\]

\( \text{AnalyzeTM} \) is the least fixed point of \( \text{AnalyzeTM}' \). \( t \) 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 \( G_s \) and a set of interference-combinations \( I \), applying \( \text{AnalyzeTM} \) to each \( G \in G_s \) 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.

\[
\text{Analyze}(G, M) = M'
\]

where \( I = \text{Interfs}(G, M) \)

\[
I' = \text{FilterFeasible}(I)
\]

\[
M' = \text{JoinMM}((\text{map}(\text{AnalyzeTM}(G), I'))
\]

\[
\text{AnalyzeAll}'(Gs, M) = \text{JoinMM}(\text{map}(\text{Analyze}(M), Gs))
\]

\[
\text{AnalyzeAll}(Gs) = \text{lp}((\text{AnalyzeAll}'(Gs), M_\bot)
\]

\text{Analyze} operates as follows: first, \( M \), the current analysis results over all threads, computes the interferences, \( I \), wrt the thread under test, \( G \). The function \text{FilterFeasible} integrates the thread-level analyzer with the feasibility analysis of Section 3.3. It expands the interferences \( I \) into a set of interference combinations \( I' \), and filters any infeasible combinations using the analysis of Section 3.3.

Specifically, given the interferences on a thread, \( I = \{(l_1 \mapsto \{e_1, e_2, \ldots\}), (l_2 \mapsto \{e_3, e_4, \ldots\}) \ldots\} \), \text{FilterFeasible} creates all combinations of pairing each load to a single interfering environment, e.g., \( I_e = \{(l_1, e_1), (l_2, e_3), \ldots\}, \{(l_1, e_2), (l_2, e_1), (l_3, e_3), \ldots\}, \ldots\). Then, \( I \) is mapped to the associated store generating the environment, e.g., \( I_s = \{(l_1, s_1), (l_2, s_3), \ldots\} \). Each set of pairs of load and store statements in \( I_s \) is then passed to the deduction analysis of Section 3.3. If it is infeasible, it is discarded, otherwise it is added to the set \( I' \) returned by \text{FilterFeasible}.

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

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

The following is a high-level example. Initially, each thread \( G \) is analyzed in the presence of \( M_\bot \), resulting in the set of interferences, \( I \), being empty (all stores map to \( \bot \)). 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 \text{AnalyzeTM} to be called once per-combination. Within a thread, the results of \text{AnalyzeTM} are joined, then, across threads, the results of \text{Analyze} are joined, creating \( M' \). The procedure repeats, thus growing the size of \( M, I \), and \( I' \) until \( M = M' \).

As in Kusano and Wang [101] 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.
Additionally, the previously described thread-level analyzer is sound for the case of verifying embedded assertions in a program [101], but not for the more general case of generating sound invariants at every program location. To generate invariants at some statement $s$, $s$ needs to be considered as a load of all shared variables such that their potential values flow to $s$. Soundness is more fully discussed in Kusano and Wang [101].

### 3.5 Experiments

We implemented our weak-memory-aware abstract interpreter in a tool named FruitTree, building upon open-source platforms such as LLVM [12], Apron [87], and μZ [79].

We implemented the current state-of-the-art MM-oblivious abstract interpretation method of Miné [121], and the SC-specific method, Watts [101]. Both are thread-modular and were implemented in the same tool as FruitTree. The analysis of Miné [121] does not include the monotonicity domain or relational lock invariants. We also compared against a previously implemented version of DUET [47, 48] (by its authors). While DUET may be unsound, and Watts is unsound, we include their results because they are closely related to our new technique.

Our analysis includes the clustering and property-directed optimizations from Kusano and Wang [101]. Clustering only considers interferences within sets of loads, similar to the packing of relational domains, and property-direction filters interference combinations unrelated to the properties under test. These optimizations reduce the number of interference combinations to test which is crucial since it grows exponentially with respect to program’s size.

Our experiments were conducted 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 [16]. The second are 52 larger applications [47, 152, 154], including Linux device drivers. The benchmarks total 61,981 lines of code. The properties under verification are expressed as assertions embedded in the program’s source code: a property is valid iff the assertion holds over all executions under a given memory model.

Our experiments 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.
3.5.1 Experiments on Litmus Tests

First, we present the litmus test results. Since these programs are small, all methods under evaluation (Miné, WATTS, DUET, and FruitTree) finished quickly. Our focus is not on comparing the runtime performance but comparing the accuracy of their results. We compare our method to these state-of-the-art techniques in terms of the number of bogus proofs 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 since we know a priori if a property holds or not.

Table 3.1: Results on the litmus test programs under TSO.

<table>
<thead>
<tr>
<th>Method</th>
<th>True Alarm</th>
<th>Bogus Alarm</th>
<th>True Proof</th>
<th>Bogus Proof</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miné [121]</td>
<td>77</td>
<td>207</td>
<td>8</td>
<td>0</td>
<td>12.9</td>
</tr>
<tr>
<td>DUET [47]</td>
<td>77</td>
<td>181</td>
<td>34</td>
<td>0</td>
<td>473.1</td>
</tr>
<tr>
<td>WATTS [101]</td>
<td>63</td>
<td>13*</td>
<td>202</td>
<td>14*</td>
<td>71.0</td>
</tr>
<tr>
<td>FruitTree</td>
<td>77</td>
<td>72</td>
<td>143</td>
<td>0</td>
<td>89.2</td>
</tr>
</tbody>
</table>

Table 3.1 summarizes the litmus test results under TSO. The first column shows the name of each method, the next four show the number of true alarms, bogus alarms, true proofs, and bogus proofs generated by each method, respectively. Since WATTS [101] was designed to be SC-specific, it ignores non-SC behaviors, its proofs are unsound under weaker memory (marked by *). The last column is the total analysis time over all tests.

Overall, the results for TSO show that the prior thread-modular technique of Miné admits many infeasible executions thus leading to 207 bogus alarms. DUET reported 177 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. The soundness of DUET under TSO or any other non-SC memory model was not explicitly discussed by its authors.

Table 3.2: Results on the litmus test programs under PSO.

<table>
<thead>
<tr>
<th>Method</th>
<th>True Alarm</th>
<th>Bogus Alarm</th>
<th>True Proof</th>
<th>Bogus Proof</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miné [121]</td>
<td>81</td>
<td>203</td>
<td>8</td>
<td>0</td>
<td>12.9</td>
</tr>
<tr>
<td>DUET [47]</td>
<td>81</td>
<td>177</td>
<td>34</td>
<td>0</td>
<td>473.1</td>
</tr>
<tr>
<td>WATTS [101]</td>
<td>64</td>
<td>12*</td>
<td>199</td>
<td>17*</td>
<td>71.0</td>
</tr>
<tr>
<td>FruitTree</td>
<td>81</td>
<td>68</td>
<td>143</td>
<td>0</td>
<td>281.4</td>
</tr>
</tbody>
</table>

Table 3.2 summarizes the results under PSO. Again, WATTS is unsound for weak-memory and cannot obtain true proofs. 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 3.3: Results on the litmus test programs under RMO.

<table>
<thead>
<tr>
<th>Method</th>
<th>True Alarm</th>
<th>Bogus Alarm</th>
<th>True Proof</th>
<th>Bogus Proof</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mné [121]</td>
<td>28</td>
<td>67</td>
<td>8</td>
<td>0</td>
<td>4.9</td>
</tr>
<tr>
<td>Duet [47]</td>
<td>11</td>
<td>58</td>
<td>34</td>
<td>0</td>
<td>187.8</td>
</tr>
<tr>
<td>Watts [101]</td>
<td>0</td>
<td>0•</td>
<td>75</td>
<td>28•</td>
<td>33.9</td>
</tr>
<tr>
<td>FruitTree</td>
<td>28</td>
<td>13</td>
<td>62</td>
<td>0</td>
<td>46.9</td>
</tr>
</tbody>
</table>

Table 3.3 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 new method was more accurate than prior techniques. However, since the analysis is over-approximated, it does not eliminate all bogus alarms. Currently, most of the 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, e.g., by extending our deductive interference feasibility analysis: we leave this as future work.

### 3.5.2 Experiments on Larger Programs

Next, we present our results on the larger benchmark programs. Since execution time is no longer negligible, we compare, across methods, both the runtime and accuracy. However, since the programs are larger (60K lines of code) and there are far too many properties, 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.

Table 3.4 shows our results under TSO. Since the results for PSO and RMO are similar we omit them for brevity. Column 1 shows the name of the benchmark program. Columns 2–3, 4–5, 6–7, and 8–9 show the runtime and number of properties verified by Mné, Duet, Watts, and FruitTree, respectively.

Again, while the proofs reported by FruitTree and Mné’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 properties proved by Mné, 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 unsound Watts “proved” 4,583 properties, possibly including some bogus proofs.

In terms of runtime, our method took 5,387 seconds, which is similar to Watts, the only
Table 3.4: Results on larger programs: ⋆ indicates unsoundly verified properties.

<table>
<thead>
<tr>
<th>Name</th>
<th>Minex [121]</th>
<th>DUET [47]</th>
<th>Watts [101]</th>
<th>FruitTree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time Verif</td>
<td>Time Verif</td>
<td>Time Verif</td>
<td>Time Verif</td>
</tr>
<tr>
<td>thread00</td>
<td>0.01 0</td>
<td>1.17 0</td>
<td>0.03 0</td>
<td>0.03 0</td>
</tr>
<tr>
<td>threadcreate01</td>
<td>0.02 1</td>
<td>0.82 1</td>
<td>0.05 2</td>
<td>0.04 2</td>
</tr>
<tr>
<td>threadcreate02</td>
<td>0.02 1</td>
<td>0.79 2</td>
<td>0.03 2</td>
<td>0.03 2</td>
</tr>
<tr>
<td>sync_01true</td>
<td>0.03 1</td>
<td>1.37 1</td>
<td>0.06 1</td>
<td>0.06 1</td>
</tr>
<tr>
<td>sync_02true</td>
<td>0.03 1</td>
<td>1.22 1</td>
<td>0.07 1</td>
<td>0.07 1</td>
</tr>
<tr>
<td>intral01</td>
<td>0.02 1</td>
<td>1.20 0</td>
<td>0.04 2</td>
<td>0.04 2</td>
</tr>
<tr>
<td>dekker01</td>
<td>3.30 3</td>
<td>1.36 2</td>
<td>6.13 4</td>
<td>6.63 4</td>
</tr>
<tr>
<td>lk2012 x2</td>
<td>0.04 3</td>
<td>1.49 3</td>
<td>0.37 4</td>
<td>0.22 4</td>
</tr>
<tr>
<td>keyISR</td>
<td>0.04 4</td>
<td>1.24 4</td>
<td>2.17 6</td>
<td>2.30 5</td>
</tr>
<tr>
<td>ib700w0d_01</td>
<td>1.72 23</td>
<td>1.89 35</td>
<td>14.5 46</td>
<td>11.4 46</td>
</tr>
<tr>
<td>ib700w0d_02</td>
<td>13.2 63</td>
<td>2.79 95</td>
<td>108.4 126</td>
<td>89.4 126</td>
</tr>
<tr>
<td>ib700w0d_03</td>
<td>23.1 81</td>
<td>2.88 122</td>
<td>178.4 162</td>
<td>156.3 162</td>
</tr>
<tr>
<td>i8xx_1c0_01</td>
<td>1.06 14</td>
<td>2.87 28</td>
<td>97.4 39</td>
<td>53.3 39</td>
</tr>
<tr>
<td>i8xx_1c0_02</td>
<td>8.57 34</td>
<td>4.69 68</td>
<td>1288.3 99</td>
<td>757.3 99</td>
</tr>
<tr>
<td>i8xx_1c0_03</td>
<td>10.7 37</td>
<td>4.73 74</td>
<td>1677.0 108</td>
<td>952.3 108</td>
</tr>
<tr>
<td>machzwd_01</td>
<td>0.49 14</td>
<td>1.34 14</td>
<td>107.5 35</td>
<td>42.1 34</td>
</tr>
<tr>
<td>machzwd_02</td>
<td>1.68 24</td>
<td>1.49 24</td>
<td>240.5 65</td>
<td>75.3 64</td>
</tr>
<tr>
<td>machzwd_03</td>
<td>4.17 39</td>
<td>1.80 39</td>
<td>488.7 110</td>
<td>128.9 109</td>
</tr>
<tr>
<td>mixcomw0d_01</td>
<td>0.81 12</td>
<td>1.96 21</td>
<td>169.3 34</td>
<td>15.3 33</td>
</tr>
<tr>
<td>mixcomw0d_02</td>
<td>2.29 32</td>
<td>4.48 41</td>
<td>768.0 64</td>
<td>61.7 63</td>
</tr>
<tr>
<td>mixcomw0d_03</td>
<td>3.56 64</td>
<td>2.93 65</td>
<td>88.3 100</td>
<td>84.0 100</td>
</tr>
<tr>
<td>pcwd_01</td>
<td>0.71 10</td>
<td>0.96 22</td>
<td>1.83 31</td>
<td>1.44 31</td>
</tr>
<tr>
<td>pcwd_02</td>
<td>4.43 27</td>
<td>1.36 56</td>
<td>8.59 82</td>
<td>6.88 82</td>
</tr>
<tr>
<td>pcwd_03</td>
<td>10.2 40</td>
<td>1.76 82</td>
<td>18.5 121</td>
<td>15.4 121</td>
</tr>
<tr>
<td>pcwd_04</td>
<td>25.4 60</td>
<td>2.16 122</td>
<td>46.1 181</td>
<td>39.8 181</td>
</tr>
<tr>
<td>sbc600xxw0d_01</td>
<td>1.11 21</td>
<td>2.02 0</td>
<td>4.96 43</td>
<td>3.09 43</td>
</tr>
<tr>
<td>sbc600xxw0d_02</td>
<td>3.31 40</td>
<td>2.31 13</td>
<td>13.7 81</td>
<td>8.04 81</td>
</tr>
<tr>
<td>sbc600xxw0d_03</td>
<td>7.04 60</td>
<td>3.29 0</td>
<td>33.4 121</td>
<td>17.9 121</td>
</tr>
<tr>
<td>sci200w0d_01</td>
<td>1.00 22</td>
<td>1.35 10</td>
<td>15.1 33</td>
<td>10.6 33</td>
</tr>
<tr>
<td>sci200w0d_02</td>
<td>6.94 58</td>
<td>2.28 24</td>
<td>64.5 87</td>
<td>47.6 87</td>
</tr>
<tr>
<td>sci200w0d_03</td>
<td>26.2 102</td>
<td>3.33 50</td>
<td>197.0 153</td>
<td>146.7 153</td>
</tr>
<tr>
<td>smsc37b787w0d_01</td>
<td>1.07 22</td>
<td>1.20 23</td>
<td>16.7 46</td>
<td>12.0 46</td>
</tr>
<tr>
<td>smsc37b787w0d_02</td>
<td>9.26 76</td>
<td>2.32 77</td>
<td>91.4 154</td>
<td>67.3 154</td>
</tr>
<tr>
<td>smsc37b787w0d_03</td>
<td>26.4 130</td>
<td>3.41 131</td>
<td>286.1 262</td>
<td>197.5 262</td>
</tr>
<tr>
<td>sc520w0d_01</td>
<td>1.59 15</td>
<td>1.15 16</td>
<td>15.6 45</td>
<td>11.0 45</td>
</tr>
<tr>
<td>sc520w0d_02</td>
<td>12.6 41</td>
<td>1.71 42</td>
<td>74.1 123</td>
<td>56.0 123</td>
</tr>
<tr>
<td>sc520w0d_03</td>
<td>27.8 58</td>
<td>2.24 59</td>
<td>155.6 174</td>
<td>116.9 174</td>
</tr>
<tr>
<td>w83877fwdt_01</td>
<td>12.5 34</td>
<td>1.82 34</td>
<td>83.9 137</td>
<td>71.0 137</td>
</tr>
<tr>
<td>w83877fwdt_02</td>
<td>29.0 50</td>
<td>2.34 50</td>
<td>189.6 201</td>
<td>159.6 201</td>
</tr>
<tr>
<td>w83877fwdt_03</td>
<td>54.2 66</td>
<td>2.70 66</td>
<td>357.9 265</td>
<td>301.9 265</td>
</tr>
<tr>
<td>wdt_01</td>
<td>0.20 3</td>
<td>1.40 13</td>
<td>65.9 14</td>
<td>27.6 14</td>
</tr>
<tr>
<td>wdt_02</td>
<td>0.39 5</td>
<td>1.54 21</td>
<td>600.2 22</td>
<td>306.5 22</td>
</tr>
<tr>
<td>wdt_03</td>
<td>0.55 6</td>
<td>1.58 25</td>
<td>1479.1 26</td>
<td>766.2 26</td>
</tr>
<tr>
<td>wdt_077_01</td>
<td>1.37 9</td>
<td>1.92 35</td>
<td>83.0 43</td>
<td>49.7 43</td>
</tr>
<tr>
<td>wdt_077_02</td>
<td>2.47 13</td>
<td>2.30 51</td>
<td>115.6 63</td>
<td>76.0 63</td>
</tr>
<tr>
<td>wdt_077_03</td>
<td>8.24 25</td>
<td>2.99 99</td>
<td>264.9 123</td>
<td>193.9 123</td>
</tr>
<tr>
<td>wdt_pci01</td>
<td>1.27 11</td>
<td>1.19 31</td>
<td>5.91 52</td>
<td>4.65 52</td>
</tr>
<tr>
<td>wdt_pci02</td>
<td>8.92 31</td>
<td>1.86 91</td>
<td>33.2 152</td>
<td>26.3 152</td>
</tr>
<tr>
<td>wdt_pci03</td>
<td>23.9 51</td>
<td>3.02 151</td>
<td>93.5 252</td>
<td>72.7 252</td>
</tr>
<tr>
<td>pcwd_pci01</td>
<td>4.72 56</td>
<td>1.38 89</td>
<td>27.2 116</td>
<td>21.1 116</td>
</tr>
<tr>
<td>pcwd_pci02</td>
<td>9.85 70</td>
<td>1.46 132</td>
<td>52.6 158</td>
<td>40.6 158</td>
</tr>
<tr>
<td>pcwd_pci03</td>
<td>20.3 88</td>
<td>2.09 186</td>
<td>97.7 212</td>
<td>72.7 212</td>
</tr>
</tbody>
</table>

Total: 415 s 1752 s 2432 s 9830 s 4583* 5387 s 4577
other flow-sensitive method, 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 a 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.

3.6 Related Work

We reviewed prior work on thread-modular abstract interpretation, which are either MM-oblivious [119–121] or SC-specific [101] in processor memory-models. We reviewed DUET [47, 48], which is non thread-modular and may be unsound under weak memory.

There are code-transformation techniques modeling weak memory [39, 100, 117], which transform a non-SC program into an SC program and then apply abstract interpretation. They generally follow the sequentialization approach pioneered by Lal and Reps [106], with a focus on code transformation as opposed to abstract interpretation. To ensure termination, they often make various assumptions to bound the program’s behavior. Furthermore, they are not thread-modular, and more importantly, they 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 modeling concurrent software on relaxed memory. They are based on either sequentializing concurrent programs [16] or encoding weak memory semantics directly using SAT/SMT solvers [15, 17]. Alglave et al. also developed techniques modeling and testing weak-memory semantics of real processors [18], and characterized the memory models of some GPUs [13]. 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, e.g., based on techniques such as stateless model checking [58, 66], a number of methods have been proposed to handle weak memory such as TSO/PSO [6, 41, 82, 171], PowerPC [9], and C++11 [129]. However, since these methods rely on concretely executing the program, and often require the user to provide test data 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.

The idea of thread-modular analysis was also explored in the model checking field [56, 59], where it was combined with predicate abstraction techniques [77], to help mitigate state
explosion and thus increase the scalability of model checkers. However, model checking is significantly different from abstract interpretation in that each thread must be abstracted into a finite-state model before it can be analyzed by finite-state model checkers. In abstract interpretation of sequential programs, Miné [118] 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. [53,54] proposed a technique for integrating 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 [88] also proposed a method for abstracting the data and control portions of a call-stack for analyzing sequential programs with potentially infinite recursion. Later, Jeannet [86] extended it to handle concurrent programs as well. However, none of these existing methods was designed specifically for handling weak memory models.

3.7 Discussion

This chapter presented a thread-modular abstract interpretation method for weak memory models, building upon a lightweight constraint system for quickly deciding the infeasibility of thread interference combinations, so they are skipped during the expensive thread analysis. The constraint system is also general enough to handle a range of processor-level memory models. We 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.
Chapter 4

Incremental Thread-Modular Abstract Interpretation

Writing correct multithreaded programs is difficult due to the astronomical number of possible interactions between threads. However, to reap the benefits of modern computer architectures multithreaded programs must be used. Automated verification techniques, such as abstract interpretation, can aid in writing multithreaded programs by certifying it is bug free.

Sadly, the same complexity confounding a programmer also hinders automated verifiers: there are far too many thread interleavings to verify individually. As a result, applying prior sequential analyses [37, 60] directly to multithreaded programs does not scale. Thread-modular techniques [102, 103, 119–121] address this by analyzing each thread in isolation thereby avoiding some of the complexity of a multithreaded program. However, even thread-modular analyses still suffer from high runtime overhead thus prohibiting their widespread use.

Coupling a change-impact analysis with a testing/verification algorithm [22, 29, 70, 105, 131] has been shown to reduce runtime overhead. The key insight of an incremental analysis is that given an old, previously verified program, $P$, and new program $P'$, we only need to verify the new behaviors in $P'$. However, there exist no prior incremental thread-modular abstract interpreters. Additionally, no prior change-impact analysis for multithreaded programs [70] was designed for weak-memory. Additionally, prior change-impact analyses for multithreaded programs [70] do not soundly model weak-memory. As such, they can only analyze programs assuming sequentially consistent memory.

In this paper we address these issues. We propose the first incremental thread-modular verification via numerical abstract interpretation. To do this, we model the semantics of weak-memory primitives within a change-impact analysis and show that the analysis is sound under various weak-memory models. Then, we integrate the change-impact information into a thread-modular analyzer to reduce runtime overhead.
Figure 4 outlines our incremental analysis. We take as input an old, previously verified program, $P$, and a new version $P'$. Via a change-impact analysis we identify statements which either may impact modified statements, or those which may be impacted by modified statements. We use these results within a thread-modular abstract interpreter to both preserve prior analysis results, and simplify the analysis of the new version by ignoring the transfer function of unnecessary statements.

Thread-modular abstract interpreters sound under weak-memory come in two categories: constraint-based [102,103], and non-constraint based [119–121]. Generally, the constraint-based analysis has both higher accuracy and runtime. Our change-impact analysis is generally applicable: we integrate it into both techniques thus developing two distinct incremental thread-modular analyzers. Furthermore, we show that both incremental analyses preserve soundness, i.e., a program verified as correct is provably bug free with respect to its specification.

The foundation of our incremental analysis is the first change-impact analysis sound under weak-memory. Prior change-impact analyses [70] for multithreaded programs considered only sequentially consistent programs, and, in particular, ignored the modeling of fences and memory-barriers. We present a change-impact analysis soundly modeling weak-memory semantics, and prove its correctness. Specifically, we introduce a semi-flow-sensitive analysis modeling the semantics of fences/memory-barriers in a partially flow-sensitive manner while still avoiding the construction of the inter-thread control-flow graph. This enables us to analyze the bulk of the program in a flow-insensitive manner thus maintaining a good balance between accuracy and runtime.

We evaluated our approach on a large set of Linux device drivers spanning 72,838 lines of code. Compared to prior non-incremental techniques [103] we reduce runtime by 82% while remaining provably sound.

To sum up, this paper makes the following contributions:

- We present a dependency analysis handling weak-memory primitives.
- We present the first change-impact analysis sound under weak-memory.
Figure 4.2: Message passing synchronization via fences.

- We integrate the change-impact analysis into a non-constraint and constraint-based thread-modular abstract interpreter to perform incremental verification.
- We evaluate our technique on a large set of Linux device drivers and show it can significantly reduce verification runtime.

### 4.1 Motivation

Next, we motivate our new approach through an example. Consider the program in Figure 4.2. Thread one writes to shared variables \( x \) and \( y \), performs a memory fence, and then writes to \( \text{flag1} \). The fence ensures the writes to \( x \) and \( y \) propagate to all other threads before the write to \( \text{flag1} \). Thread two similarly performs a write to \( z \), a fence, and a write to \( \text{flag2} \).

Thread three upon reading the value of \( \text{flag1} \) to be true asserts \( x + y == 15 \). This assertion holds since the writes to \( x \) and \( y \) of 5 and 10 must have occurred before \( \text{flag1} \) is set to 1. Similarly, thread three also asserts that the value of \( z \) is 1 when \( \text{flag2} \) is true.

Prior thread-modular abstract interpreters [103] can verify the properties in Figure 4.2. They do this by generating interferences, or stores to shared memory, from threads one and two upon thread three, and then analyze thread three in the presence of these interferences.

Consider, however, that thread one was modified as shown in Figure 4.3. The fence changed to a store–store memory barrier, i.e., an instruction ensuring all stores before the barrier are ordered with all subsequent stores. The assertions in thread three still hold since the stores to \( x \) and \( y \) are guaranteed to be visible to thread three before the write to \( \text{flag1} \). However, existing thread-modular analyzers [102, 103, 119–121] would re-analyze the entire program even though only a subset, the interactions between thread one and thread three, are impacted by the modification. We would like an automated technique to identify impacted statements from a modification, and allow the thread-modular analyzer to consider only the new program behavior.

Designing an incremental thread-modular analyzer capable of reducing the runtime overhead of analyzing programs such as Figure 4.2 poses two challenges. First, existing change-impact
analyses for concurrent programs [70] do not handle fences and memory-barriers. To solve this, we soundly model the semantics of weak-memory primitives within a dependency analysis allowing a weak-memory aware change-impact analysis to be calculated. And, second, it is unclear how to soundly integrate a change-impact analysis into a thread-modular analyzer. We show we can preserve portions of the analysis results from the prior version into the new version, and also can ignore the transfer function of non-impacted statements: both of these, as we show experimentally, can significantly reduce runtime.

Next we illustrate these two solutions on Figure 4.2. The dependency analysis results can be seen in Figure 4.4. An edge $n_0 \rightarrow n_1$ indicates that the computation of $n_1$ depends on $n_0$. The modified statement, the red solid-rectangular node, only impacts a small portion of thread three. We separately identify statements which may-impact the modified statement (blue and dotted), and those which are impacted by the modification (green and dashed). Statements which neither may-impact nor are impacted by changes are in rounded solid black.

Categorizing statements as either may-impact or impacted is done by first running a dependency analysis, and then performing a change-impact analysis. Our dependency analysis specifically handles weak-memory primitives such as fences and memory-barriers: ignoring these primitives would make the analysis unsound, since, e.g., the dependencies from the memory-barrier to thread-three would be excluded. Then, we integrated the dependency analysis into a change-impact analysis to identify affected code regions, as seen in Figure 4.4.

We make two modifications to the abstract interpreter to boost runtime performance. First, statements which either only may-impact or were unaffected by modifications can preserve their analysis results from the old version. Intuitively, this can be done because the statements are not impacted: they were unaffected by modifications so their results will remain the same. Second, unaffected statements (neither impacted nor may-impact) can use the identity function as their transfer function, i.e., they pass the memory-state of their predecessor(s) to their successors. This is because their computation will not affect any modified code regions. Using the identity function can offer significant speedups since it skips expensive operations within the abstract domain in use. In what remains, we describe how these analyses are performed, and why they are sound.
4.2 Thread-Modular Abstract Interpretation

In this section, we introduce first the non-constraint based thread-modular abstract interpreter, and then one with constraints. More in-depth discussions can be found in [102,119–121].

Thread-modular abstract interpretation analyzes each thread in isolation, as if it were a sequential program, and then propagates the interferences on shared memory across threads. As a result, the thread-modular analysis—contrast to an analysis of the full concurrent control-flow graph—is more scalable: the complexity of the concurrent control-flow graph, i.e., the composition of control-flow graphs of all threads, is exponential with respect to the number of threads.

4.2.1 Preliminaries

A program $P$ is a finite set of threads where each $T \in P$ is a control-flow graph, i.e., a tuple $(N, n_0, \delta)$ where $N$ is the set of nodes in $T$, $n_0 \in N$ the entry of $T$, and $\delta \subseteq N \times N$ is the transition-relation; i.e., $(n, n') \in \delta$ indicates that control may flow from node $n$ to $n'$ in $T$.

We associate each node $n$ with an abstract memory state $s$. Such states can be represented in abstract domains such as octagons [118], or intervals [37]. We assume the domains are a lattice with minimal (bottom, ⊥) and maximal (top, ⊤) elements, a partial-order relation ($\sqsubseteq$), and a join operator ($\sqcup$).
Consider the following example of abstract domains. The value of a variable in the interval domain is represented by its lower and upper bounds, e.g., a variable \( x \) could be \([-5, 5]\) representing all values between \(-5\) and \(5\). Joining two intervals \( i_1 \sqcup i_2 \) is the interval containing both \( i_1 \) and \( i_2 \), e.g., \([0, 5] \sqcup [7, 10] = [0, 10]\). The two intervals are partially-ordered, \( i_1 \sqsubseteq i_2 \) if the interval of \( i_1 \) is contained in \( i_2 \), e.g., \([4, 5] \sqsubseteq [0, 10]\) and \([1, 2] \not\sqsubseteq [100, 200]\). An abstract memory state in the interval domain maps each variable to an interval, e.g., in a program with two variables \( x \) and \( y \) a potential memory state is \((x \mapsto [0, 0], y \mapsto [-5, 5])\). \( \sqcup \) and \( \sqsubseteq \) are defined over states by applying the operators for individual intervals on each variable.

Additionally, each node \( n \) is associated with a transfer function. \( n \)'s transfer function takes a memory state \( s \) and produces a memory state \( s' \) over-approximating the memory-state after \( n \). The function \( \text{tfunc} \) takes a node \( n \) and a memory state \( s \), and returns the result of executing \( n \) with input \( s \).

Consider the following example of transfer functions. Let \( s = (x \mapsto [0, 0], y \mapsto [-5, 5]) \) be the state before the statement \( x = y + 1 \). Let \( tf \) be the transfer function for the statement, then \( tf(s) = (x \mapsto [-4, 6], y \mapsto [-5, 5]) \).

Finally, the thread-modular analysis introduces interferences. An interference maps each variable in the program to an interval\(^1\). At a high level, an interference represents the propagation of shared memory effects across threads. To do this, during the abstract interpretation of thread \( T \) the interference \( I \) contains all values stored into shared memory from threads other than \( T \). Then, when \( T \) performs a shared memory read of variable \( v \) it can either read the value from its own thread-local memory state or from the value in the interference.

\[\text{Algorithm 5} \text{ Analysis of an individual thread } T \text{ in the presence of interference } I.\]

1: \textbf{function} \text{AnalyzeThread}(T = \langle N, n_0, \delta \rangle, I) \hfill \\
2: \quad \triangleright \text{Initialize } S \text{, a map from nodes to states} \hfill \\
3: \quad S(n_0) \leftarrow \top, \text{ otherwise } S(n) \leftarrow \bot \hfill \\
4: \quad W \leftarrow \{n_0\} \hfill \triangleright W \text{ is a set of nodes to process} \hfill \\
5: \quad \textbf{while } \exists n \in W \hfill \\
6: \quad \quad W \leftarrow W \setminus \{n\} \hfill \\
7: \quad \quad \textbf{if } n \text{ is a shared-memory read of variable } v \hfill \\
8: \quad \quad \quad s \leftarrow \text{tfunc}(n, S(n) \cup I(v)) \hfill \\
9: \quad \quad \textbf{else} \hfill \\
10: \quad \quad \quad s \leftarrow \text{tfunc}(n, S(n)) \hfill \\
11: \quad \quad \textbf{for all } \langle n, n' \rangle \in \delta \text{ such that } s \not\sqsubseteq S(n') \hfill \\
12: \quad \quad \quad S(n') \leftarrow S(n') \cup s \hfill \\
13: \quad \quad \quad W \leftarrow W \cup \{n'\} \hfill \\
14: \quad \textbf{return } S \hfill \\

\(^1\)More complex definitions of interferences exist \([102, 119–121]\), for simplicity of presentation we assume they are as defined here.
4.2.2 Thread-Modular Abstract Interpretation Without Constraints

Algorithm 5 analyzes a single thread $T$ in the presence of interference $I$. Specifically, for a variable $v$, $I(v)$ are the values stored into $v$ from threads other than $T$. The procedure is sequential abstract interpretation except for the use of $I$ when encountering a load. The algorithm returns a map $S$ from each node in $T$ to its memory state. The set $W$ contains nodes to be processed, initially set to $n_0$. Upon removal of an $n \in W$, if $n$ is a load then the input state to $n$ ($S(n)$) is joined with the interference on $v$ ($I(v)$). Otherwise, the input state for $n$ is simply that in $S$. Finally, any successor of $n$ in $T$ has its input updated and is added to $W$ if the result of the transfer-function of $n$ has not reached a fixed point.

**Algorithm 6** Analysis of a program $P$, i.e., a set of threads.

1: function AnalyzeProgram($P$)  
2:    $Ss \leftarrow$ map all nodes in $P$ to $\bot$  
3:    $Ss' \leftarrow Ss$  
4: repeat  
5:    $Ss = Ss'$  
6:    for all $T \in P$  
7:        $I \leftarrow \biguplus \text{Interf}(T', Ss)$ for each $T' \in P, T' \neq T$  
8:        $Ss' \leftarrow Ss' \uplus \text{AnalyzeThread}(T, I)$  
9: until $Ss = Ss'$  
10: $\triangledown$ Return interference of thread $T$ given analysis results $Ss$  
11: function Interf($T = (N,n_0,\delta), Ss$)  
12:    $I \leftarrow \emptyset$  
13:    for all $n \in N$  
14:        if $n$ is a shared memory write to variable $v$  
15:            $I(v) \leftarrow I(v) \uplus \text{TFUNC}(n, Ss(n))$  
16:    return $I$

Next, Algorithm 6 analyzes a set of threads using the previous analysis of a single thread. For each thread $T$ in the program $P$ repeatedly calculate the interferences upon $T$ and analyze it in isolation until a fixed point is reached. Specifically, $\text{Interf}$ joins the memory-state after each store the within some thread $T'$. We use $\uplus$ to be the join of the memory states on matching variables/nodes within a map. So, on line 7, $I(v)$ contains the join of all states after writes to $v$ from threads other than $T$. Next, we use $\text{AnalyzeThread}$ (Algorithm 5) to analyze $T$ in the presence of $I$. The results are joined into $Ss$. The process continues until the analysis results, and thus the interferences, reach a fixed point. Algorithm 6 is sound under TSO, PSO, and RMO, since it permits all orderings of loads and stores. For details see [121].

Next, we exemplify the previous Algorithms on Figure 4.2. Initially, $Ss$ maps all nodes to $\bot$. Since $Ss$ maps all nodes to $\bot$, the interferences also are all $\bot$. Thus, analyzing each thread is equivalent to its analysis as a sequential program: the join with the interferences on line 8 in $\text{AnalyzeThread}$ is always $S(n) \uplus \bot \equiv S(n)$. The results of analyzing each thread in isolation are joined into $Ss'$ mapping each node in the program to the abstract memory-state
Algorithm 7 Flow-sensitive thread-modular analysis.

1: function AnalyzeProg-Flow($P$: a set of CFGs)
2: $Ss \leftarrow \emptyset$
3: $I \leftarrow \emptyset$
4: repeat
5: $I' \leftarrow I$
6: for all $T \in P$
7: $I_c \leftarrow \text{InterferenceComboFeasible}(T, I)$
8: for all $i_c \in I_c$
9: $\text{Env} \leftarrow \text{AnalyzeThread-Mod}(T, i_c)$
10: $Ss \leftarrow Ss \cup \text{Env}$
11: $I \leftarrow \text{Interfs-Flow}(T, Ss, I)$
12: until $I = I'$
13: return $Ss$
14:
15: function Interfs-Flow($T, Ss, I$)
16: for all $(n, e) \in Ss$
17: if $n$ is a shared memory write in $T$
18: $I(T) \leftarrow I(T) \cup \{(n, \text{tfunc}(n, e))\}$
19: return $I$
20:
21: function InterferenceComboFeasible($T, I$)
22: $I_c \leftarrow \emptyset$
23: $\text{VEs} \leftarrow \{(n, e) | (n, e) \in I(T'), T' \in P, \text{ and } T' \neq T\}$
24: for all $l \in \text{Loads}(T)$
25: $\text{LIs}(l) \leftarrow \{(s_{\text{dummy}}, e_{\text{self}})\}$
26: if $l$ is not self-reachable
27: for all $(n, e) \in \text{VEs}$
28: if $\text{LoadVar}(l) = \text{StoreVar}(n)$
29: $\text{LIs}(l) \leftarrow \text{LIs}(l) \cup \{(n, e)\}$
30: else \hspace{1cm} \triangleright \text{Handling loads in loops}
31: for all $(n, e) \in \text{VEs}$
32: if $(\text{LoadVar}(l) = \text{StoreVar}(n))$
33: and $\neg \text{MustHappenBefore}(l, n)$
34: $\text{LIs}(l) \leftarrow \text{LIs}(l) \cup \{(s_{\text{dummy}}, e)\}$
35: $\text{Es} \leftarrow \text{CartesianProduct}($\text{LIs}$)
36: for all $i_c \in \text{Es}$
37: if $\text{QUERY.IsFeasible}(i_c)$
38: $I_c \leftarrow I_c \cup \{i_c\}$
39: return $I_c$

before its execution.

On the next iteration in $\text{ANALYZEProg}$ (lines 4–9), the interferences are non-empty. For example, when analyzing thread 3 the interferences are: $I = \langle x \mapsto [5, 5], y \mapsto [10, 10], \text{flag1} \mapsto [1, 1], \text{flag2} \mapsto [1, 1] \rangle$. The interferences on thread 1 and 2 are irrelevant since they do not perform loads. Then, on line 8 of $\text{ANALYZEThread}$ the read of $\text{flag1}$ in thread 3 loads $[0, 1]$ containing either the thread-local value $[0, 0]$ or the value interfering value $[1, 1]$). So, the first branch in thread 3 is taken and the read of $x$ loads $[0, 5]$ (thread-local value $[0, 0]$, or interference $[5, 5]$.); $y$ similarly loads $[0, 10]$. Thus, the assertion is incorrectly violated since $x + y = [0, 15]$. The second assertion is similarly violated.
4.2.3 Thread-Modular Abstract Interpretation With Constraints

The previous analysis is not flow-sensitive across threads: a load \( l \) of \( v \) may observe \textit{any} value stored into \( v \) This includes values, as in Figure 4.2, from a store \( s \) where there is no concrete execution where \( l \) observes \( s \). While we would like to maintain all control information across threads, thus being fully flow-sensitive, this is prohibitively expensive. Accuracy-wise, we would like to be fully flow-sensitive and maintain all control information across threads, but this is prohibitively expensive. Thus, we need a middle ground maintaining some flow-sensitivity while still thread-modular.

Algorithm 7 [102] addresses these issues in two ways: first, rather than joining all interferences into a single value it analyzes each \textit{interference combination} individually. An interference combination maps each load to a single interfering store. Second, it queries the feasibility of each interference combination using a lightweight Datalog analysis. In the remainder of this section we discuss both points.

The function AnalyzeProg-Flow analyzes a set \( P \) of threads similar to Algorithm 6. \( I \) maps each thread \( T \in P \) to its interferences, i.e., each pair \((n, s) \in I(T)\) indicates that \( s \) is the state after some store \( n \) in \( T \). Each thread is analyzed until the interferences reach a fixed point (lines 4–12).

The function InterferenceComboFeasible returns a set, \( I_c \) of feasible interference combinations (line 7). \( T \) is analyzed in the presence of each \( i_c \in I_c \) pairing a load with an interfering store and state. The function AnalyzeThread-Mod, omitted for brevity, is AnalyzeThread except the second argument, the interference, pairs each load \( l \) to the interference, \((s, e)\), it should read from. The function Interfs-Flow updates the interferences generated by \( T \).

Next, we discuss InterferenceComboFeasible in more detail. \( I_c \) is the set of interference combinations relevant to a thread, \( T \). First, the set of pairs of interferences (stores and states) from threads other than \( T \) is stored into \( VEs \) (line 23). Next, each load \( l \) in \( T \) is mapped to the set of store–state pairs which may interfere with \( l \) (lines 24–33). Next, \( LIs \) maps each load \( l \) to a set of interfering store–state pairs \( l \) (lines 24–33). First, \( l \) is mapped to the pair, \((s_{\text{dummy}}, e_{\text{self}})\), indicating the load should read from its local state. Second, we pair each non-self-reachable load with all interfering stores from \( VEs \) (lines 26–29). A self-reachable load \( l \) is paired with the join of all stores running in parallel with \( l \) (lines 30–33). Next, the interference combinations are created from the Cartesian product of \( LIs \), i.e., if \( LIs \) contains \( k \) items the function CartesianProduct (line 34) returns the set \( LIs(l_1) \times \cdots \times LIs(l_k) \). Finally, for each \( i_c \in I_c \) we check if the combination must not be feasible via a constraint system.

For clarity we describe the application of InterferenceComboFeasible to a simplified version of Figure 4.2 in Figure 4.5. Thread 2 has loads \( l_1 \) and \( l_2 \) to variables \( y \) and \( x \), and thread 1 has interfering stores \( s_1 \) and \( s_2 \), to \( x \), and \( s_3 \) to \( y \). Let the state after each store be \( e_1 \), \( e_2 \), and \( e_3 \), respectively. First \( VEs \) contains all pairs of interfering stores and their
int x = y = 0;
void thread1() {
  x = 1;  // s1
  x = 5;  // s2
  fence;
  y = 10; // s3
}

Figure 4.5: Example for the constraint-based abstract interpreter.

The final portion of Algorithm 7, the constraint analysis, determines and removes any infeasible combination from \( L_c \). Here, infeasible indicates that the interference combination could not happen in any concrete execution. The analysis is implemented and specified declaratively. Conveniently, the infeasibility analysis handles sequential consistency [107] (SC), as well as TSO [142, 161], PSO [148], and RMO [161].

The analysis encodes the interference combination under test, \( i_c \), along with the program-order constraints. These constraints state which statements within the program may be re-ordered [11] and are governed by both the program and the memory model. For example, under TSO the statements \( x = 5; \ t1 = y \) may be reordered. Under PSO, \( x = 10; \ y = 20 \) can also be reordered.

First, we define the relations used in the analysis. For a relation \( R \) we use \( R(x_1, \ldots) \) to mean \( (x_1, \ldots) \in R \).

- **LOAD** \((l, v)\) \( l \) is a load of \( v \).
- **STORE** \((s, v)\) \( s \) is a store to \( v \).
- **LLMembar** \((s)\) \( s \) is a load–load barrier (similarly, **LSMembar**, **SLMembar**, and **SSMembar** for load–store, store–load, and store–store barriers).
- **DOMINATES** \((s_1, s_2)\) \( s_1 \) dominates \( s_2 \) in some thread.
- **NOTREACHABLEFROM** \((s_1, s_2)\) \( s_1 \) cannot be reached from \( s_2 \) within a thread.
- **THREADCREATES** \((s_1, s_2)\) \( s_1 \) creates child thread with entry \( s_2 \).
- **THREADJOINS** \((s_1, s_2)\) \( s_1 \) is the join of a thread with last operation \( s_2 \).
- **MHB** \((s_1, s_2)\) \( s_1 \) must happen before \( s_2 \).

state, \( VEs = \{(s_1, e_1), (s_2, e_2), (s_3, e_3)\} \). Next, \( LIIs \) pairs \( l_1 \) and \( l_2 \) with all interferences in \( VEs: LIIs(l_1) = \{(s_3, e_3), (s_{dummy}, e_self)\} \) and \( LIIs(l_2) = \{(s_1, e_1), (s_2, e_2), (s_{dummy}, e_self)\} \). The Cartesian product of \( LIIs \) is the set of interference combinations:

- \( i_1 = \{(l_1, (s_{dummy}, e_self)), (l_2, (s_{dummy}, e_self))\} \)
- \( i_2 = \{(l_1, (s_{dummy}, e_self)), (l_2, (s_1, e_1))\} \)
- \( i_3 = \{(l_1, (s_{dummy}, e_self)), (l_2, (s_2, e_2))\} \)
- \( i_4 = \{(l_1, (s_3, e_3)), (l_2, (s_{dummy}, e_self))\} \)
- \( i_5 = \{(l_1, (s_3, e_3)), (l_2, (s_1, e_1))\} \)
- \( i_6 = \{(l_1, (s_3, e_3)), (l_2, (s_2, e_2))\} \)

The analysis encodes the interference combination under test, \( i_c \), along with the program-order constraints. These constraints state which statements within the program may be re-ordered [11] and are governed by both the program and the memory model. For example, under TSO the statements \( x = 5; \ t1 = y \) may be reordered. Under PSO, \( x = 10; \ y = 20 \) can also be reordered.
ReadsFrom\((s_1, s_2)\) $s_1$ loads the value from store $s_2$.

MustNotReadFrom\((s_1, s_2)\) $s_1$ must not read from $s_2$.

We use $\_\_$ when we do not care about a variable, e.g., $\text{Store}(s, \_\_)$. The first ten relations ($\text{Load}$, $\text{Store}$, $\text{LLMembar}$, $\text{LSMembar}$, $\text{SLMembar}$, $\text{SSMemBar}$, $\text{Dominates}$, $\text{NotReachableFrom}$, $\text{ThreadCreates}$, $\text{ThreadJoins}$) come directly from each thread’s control-flow graph. MHB is calculated initially from the first ten. ReadsFrom encodes the interference combination under test. The goal of the analysis is to calculate the relation MustNotReadFrom. Any $(s_1, s_2) \in \text{MustNotReadFrom}$ while simultaneously $(s_1, s_2) \in \text{ReadsFrom}$ indicates the interference combination under analysis is infeasible.

Next, we introduce the deduction rules for the analysis. First, the relation $\text{NoReorder}(s_1, s_2)$ captures when two statements $s_1$ and $s_2$ cannot be reordered.

Under SC, it is defined as:

\[
\top \quad \text{NoReorder}(s_1, s_2) \quad \text{under SC}
\]

That is, under SC no statements may be reordered.

For TSO, $\text{NoReorder}$ is defined as:

\[
\text{Load}(l, \_\_) \quad \text{NoReorder}(l, st) \quad \text{under TSO} \\
\text{Store}(s, \_\_) \quad \text{NoReorder}(st, s) \quad \text{under TSO}
\]

A load cannot be reordered with any subsequent statement, and a store cannot be reordered with any preceding statement.

For PSO, $\text{NoReorder}$ is defined as:

\[
\text{Load}(l, \_\_) \quad \text{NoReorder}(l, st) \quad \text{under PSO} \\
\text{Store}(s_1, v) \land \text{Store}(s_2, v) \quad \text{NoReorder}(s_1, s_2) \quad \text{under PSO}
\]

A load cannot be reordered with any subsequent statement, and two stores to the same variable cannot be reordered.

For RMO, $\text{NoReorder}$ is defined as:

\[
\text{Load}(s_1, v_1) \land \text{Load}(s_2, v_1) \quad \text{NoReorder}(s_1, s_2) \quad \text{under RMO} \\
\text{Load}(s_1, v_1) \land \text{Store}(s_2, v_1) \quad \text{NoReorder}(s_1, s_2) \quad \text{under RMO} \\
\text{Store}(s_1, v_1) \land \text{Store}(s_2, v_1) \quad \text{NoReorder}(s_1, s_2) \quad \text{under RMO}
\]

These rules can be interpreted as with TSO and PSO.
Fences and memory-barriers induce additional ordering constraints, e.g., a load–store memory barrier prevents loads occurring before the barrier being reordered with subsequent stores. These are also modeled with \( \text{NoReorder} \).

\[
\begin{align*}
\text{LLMembar}(m) \land \text{Load}(s_1, \_ ) \land \text{Load}(s_2, \_ ) \\
\land \text{Dominates}(s_1, m) \land \text{Dominates}(m, s_2) \\
\text{NoReorder}(s_1, s_2) \\
\text{LSMembar}(m) \land \text{Load}(s_1, \_ ) \land \text{Store}(s_2, \_ ) \\
\land \text{Dominates}(s_1, m) \land \text{Dominates}(m, s_2) \\
(s_1, s_2) \in \text{NoReorder} \\
\text{SLMembar}(m) \land \text{Store}(s_1, \_ ) \land \text{Load}(s_2, \_ ) \\
\land \text{Dominates}(s_1, m) \land \text{Dominates}(m, s_2) \\
\text{NoReorder}(s_1, s_2) \\
\text{SSMembar}(m) \land \text{Store}(s_1, \_ ) \land \text{Store}(s_2, \_ ) \\
\land \text{Dominates}(s_1, m) \land \text{Dominates}(m, s_2) \\
\text{NoReorder}(s_1, s_2)
\end{align*}
\]

A fence is semantically all four types of barriers:

\[
\begin{align*}
\text{Fence}(f) \\
\text{LLMembar}(f) \\
\text{Fence}(f) \\
\text{LSMembar}(f) \\
\text{Fence}(f) \\
\text{SLMembar}(f) \\
\text{Fence}(f) \\
\text{SSMembar}(f)
\end{align*}
\]

Synchronization routines, such as \texttt{lock/unlock} and compare-and-swap implicitly use fences/memory-barriers e.g., to ensure data-race freedom [10]. This can be modeled using the previous by making each routine an appropriate fence.

Next, using the previous relations we calculate the must-happen-before relation \( \text{MHB} \). First, thread creation join trivially induce \( \text{MHB} \) membership. Given a thread-creation/thread-join site \( s \) and thread with entry statement \( s_e \), and exit \( s_x \):

\[
\begin{align*}
\text{ThreadCreates}(s, s_e) \\
\text{MHB}(s, s_e) \\
\text{ThreadJoins}(s, s_x) \\
\text{MHB}(s_x, s)
\end{align*}
\] (4.1)

Next, we use \( \text{Dominates}(s_1, s_2) \) to determine if \( s_1 \) occurs before \( s_2 \). Thus, if \( \text{NoReorder} \) prevents the reordering then \( \text{MHB}(s_1, s_2) \). Additionally, \( s_1 \) must not be not self-reachable. In this case, \( s_1 \) may occur both before and after \( s_2 \) so no \( \text{MHB} \) membership is deduced. Formally:

\[
\begin{align*}
\text{Dominates}(s_1, s_2) \land \text{NotReachableFrom}(s_2, s_1) \\
\land \text{NoReorder}(s_1, s_2) \\
\text{MHB}(s_1, s_2)
\end{align*}
\] (4.2)

Additionally, \( \text{MHB} \) is transitive:

\[
\begin{align*}
\text{MHB}(s_1, s_2) \land \text{MHB}(s_2, s_3) \\
\text{MHB}(s_1, s_3)
\end{align*}
\] (4.3)
Lastly, we deduce MHB membership using the interference combination under test, i.e., \textsc{ReadsFrom}. We capture the case of \( t_1 = x \) (a load of \( x \)) running in parallel with \( x = 5; x = 10 \) (two stores into \( x \)). If \( t_1 \) reads 5 then the load must-happen-before \( x = 10 \). Otherwise, the second store would have overwritten the value read by \( t_1 \). Formally:

\[
\begin{align*}
\text{ReadsFrom}(l, s_1) & \land \text{MHB}(s_1, s_2) \\
\land \text{Load}(l, v) & \land \text{Store}(s_1, v) \land \text{Store}(s_2, v) \\
\hline
\text{MHB}(l, s_2)
\end{align*}
\]

(4.4)

Since the rule depends on \textsc{ReadsFrom}, i.e., the interference combination, it deduces MHB membership only valid assuming the interference combination, not over all executions.

Finally, we deduce \textsc{MustNotReadFrom}. Intuitively, \( \text{MHB}(l, s) \) implies \( l \) cannot read from \( s \) since \( s \) has not occurred.

\[
\begin{align*}
\text{MHB}(l, s) \\
\hline
\text{MustNotReadFrom}(l, s)
\end{align*}
\]

(4.5)

The final rule captures \( t_1 = x; x = 10; t_2 = x \) running in parallel with \( x = 5 \). If \( t_1 \) reads from \( x = 5 \) then \( t_2 \) must not also read 5 since the store \( x = 10 \) must have overwritten the value. Formally:

\[
\begin{align*}
\text{ReadsFrom}(l_1, s_1) & \land \text{MHB}(l_1, s_2) \land \text{MHB}(s_2, l_2) \\
\land \text{Load}(l_1, v) & \land \text{Load}(l_2, v) \land \text{Store}(s_2, v) \\
\hline
\text{MustNotReadFrom}(l_2, s_1)
\end{align*}
\]

(4.6)

At this point we presented the constraint analysis: given a program and an interference combination it determines if the interference combination is infeasible. This is integrated into the thread-modular analysis in Algorithm 7 line 36. It is sound since each of the rules in isolation is sound. Its integration with the abstract interpreter is a form of semantic reduction [37] and is thus also sound. See [102,103] for more.

Next, we revisit the example in Figure 4.5. First, the program is safe under SC, TSO, PSO, and RMO: thread 2 reading \( y \) as 10 implies the store \( x = 5 \) in thread 1 occurred. Recall, Algorithm 7 considers six combinations, \( i_1 \)–\( i_6 \), for thread 2. A thread-modular analysis without interference feasibility checking incorrectly reports a false alarm since the ordering of stores in thread 1 is lost [119–121] (as in Figure 4.2). Specifically, combination \( i_5 \) causes an error.

Next, we show the analysis operating on \( i_5 \), corresponding to \( y \) reading 10 and \( x \) reading 1. The combination is encoded as \textsc{ReadsFrom}(\( l_1, s_2 \)) and \textsc{ReadsFrom}(\( l_2, s_1 \)). First, in all the considered memory models the statements in thread one must be in program order, i.e., \( \text{MHB}(s_1, s_2) \), \( \text{MHB}(s_2, s_3) \), and \( \text{MHB}(s_1, s_3) \). This is deduced via Rules 4.2 and 4.3. \textsc{NoReorder} prevents this reordering due to the SC/TSO specific rules, and due to the fence under PSO/RMO. Similarly, we have \( \text{MHB}(l_1, l_2) \). From Rule 4.4, \( \text{MHB}(l_2, s_2) \): \( s_2 \) executing before \( l_2 \) would overwrite the value written by \( s_1 \). Via the transitive property we have \( \text{MHB}(l_2, s_3) \), and \( \text{MHB}(l_1, s_3) \). Thus, \textsc{NotReadFrom}(\( l_1, s_3 \)) by Rule 4.5. We’ve reached a contradiction since \( (l_1, s_3) \) is simultaneously in \textsc{ReadsFrom} and \textsc{NotReadFrom}, thus the combination is infeasible.
4.3 Calculating Dependencies Under Weak Memory Models

Next, we present a dependency analysis sound for programs written under weak memory. The main challenge is accurately modeling the semantics of fence and memory barrier instructions. To do so, we introduce a semi-flow-sensitive analysis.

A fully flow-sensitive analysis for concurrent programs requires an accurate inter-thread control-flow graph, i.e., the composition of each thread’s control-flow graph accounting for inter-thread synchronization. Since constructing the inter-thread control-flow graph is prohibitively expensive we would like to use a more efficiently computable flow-insensitive analysis. A flow-insensitive analysis can often be sufficiently accurate when analyzing a program in static-single assignment form since it inherently captures the control-flow of the program. But, as we will show, the nature of fences and memory barriers requires some reasoning about the program’s control-flow. The question then is how to inject sufficient flow-sensitivity to the analysis in order to reason about their semantics.

Typically, a flow-insensitive dependency analysis considers the semantics of each instruction individually, e.g., an add instruction, $a = x + y$, indicates that $a$ is dependent on both $x$ and $y$. However, the semantics of fences and memory barriers are not immediately obvious for two reasons: first, they do not have explicit operands. And second, their semantics depends on the ordering of statements in the program. For example the program in Figure 4.5 uses a fence to prevent the reordering of writes to $x$ and $y$ in thread one. Semantically, the fence flushes a thread’s store-buffers so there are store-to-fence dependencies (the store impacts the flushed value). Similarly, the load in thread two may read the flushed value so there is a fence-to-load dependency. The full dependency graph is in Figure 4.6. The fence related dependencies can only be determined by knowing the stores to $x$ occur before the fence. This differs greatly from the traditional flow-insensitive analysis: each statement cannot be analyzed in isolation.

The analysis of fences could be done flow-insensitively if we, conceptually, make a fence/memory-barrier a read/write of all global variables. However, this causes many spurious dependencies since threads usually synchronize on subsets of all global variables. For example, in Figure 4.2 this causes dependencies between all three threads which is too inaccurate to reason about the branches in thread 3 independently.

Next, we calculate fence/barrier using a system of constraints. As in the constraint analysis of Section 4.2 we use a set of input relations, and define a set of rules generating new relations. Repeatedly applying the rules until a fixed point is reached calculates the data-dependency relation.

Our analysis uses the relation $\text{Reachable}(a, b)$ as input, indicating statement $b$ is reachable from $a$ within some thread’s control-flow graph. Also, it uses the relations $\text{LLMemBar}$, $\text{LSMemBar}$, $\text{SLMemBar}$, $\text{SSMemBar}$, $\text{Fence}$, $\text{Store}$, and $\text{Load}$ as defined in Sec-
Next, we present the rules computing dependencies involving fences/memory-barriers within a thread.

\[
\begin{align*}
&\text{SLMemBar}(m) \land \text{Store}(s, \_ ) \land \text{Reachable}(s, m) \\
&\quad \Rightarrow \text{DATADep}(s, m) \\
&\text{SSMemBar}(m) \land \text{Store}(s, \_ ) \land \text{Reachable}(s, m) \\
&\quad \Rightarrow \text{DATADep}(s, m) \\
&\text{LSMemBar}(m) \land \text{Load}(l, \_ ) \land \text{Reachable}(l, m) \\
&\quad \Rightarrow \text{DATADep}(l, m) \\
&\text{LLMemBar}(m) \land \text{Load}(l, \_ ) \land \text{Reachable}(l, m) \\
&\quad \Rightarrow \text{DATADep}(l, m) \\
&\text{SLMemBar}(m) \land \text{Load}(l, \_ ) \land \text{Reachable}(m, l) \\
&\quad \Rightarrow \text{DATADep}(m, l) \\
&\text{SSMemBar}(m) \land \text{Store}(s, \_ ) \land \text{Reachable}(m, s) \\
&\quad \Rightarrow \text{DATADep}(m, s) \\
&\text{LSMemBar}(m) \land \text{Store}(s, \_ ) \land \text{Reachable}(m, s) \\
&\quad \Rightarrow \text{DATADep}(m, s) \\
&\text{LLMemBar}(m) \land \text{Load}(l, \_ ) \land \text{Reachable}(m, l) \\
&\quad \Rightarrow \text{DATADep}(m, l)
\end{align*}
\]

The analysis computes the \text{DATADep} relation where \text{DATADep}(a, b) indicates \(b\) is data-dependent on \(a\). Equations 4.7a–4.7h define the dependencies between stores/loads before/after the various types of barriers. The equations only consider dependencies within a thread since \text{Reachable} is from each individual thread’s control-flow graph. For example, Equation 4.7a creates a dependency from a store \(s\) occurring before a store–load barrier \(m\). Similarly, Equation 4.7e considers a load after a store–load barrier. From a programmers perspective, these rules prevent the hardware/compiler from reordering seemingly independent statements (e.g., \(t1 = x; \ t2 = y\)).

Next, we define dependencies across threads.

\[
\begin{align*}
&\text{DATADep}(s, m) \land \text{Store}(s, v) \land \text{SLMemBar}(m) \land \text{Load}(l, v) \\
&\quad \Rightarrow \text{DATADep}(m, l)
\end{align*}
\]
\[
\frac{\text{DataDep}(s, m) \land \text{Store}(s, v) \land \text{SSMemBar}(m) \land \text{Load}(l, v)}{\text{DataDep}(m, l)}
\] (4.8b)

Equations 4.8a and 4.8b operate across threads since the Reachable relation is not used. Inter-thread reachability could be used but this requires an accurate inter-thread CFG. Equation 4.8a considers a store \( s \) to variable \( v \) before a store–load barrier \( m \). This induces a barrier–load dependency between \( m \) and a subsequent load of \( v \) since the \( m \) flushes \( s \)’s write of \( v \) to shared-memory. \( s \) occurs before \( m \) since DataDep\((s, m)\), as in Equation 4.7a requiresReachable\((s, m)\).

Finally, data dependencies are transitive:
\[
\frac{\text{DataDep}(a, b) \land \text{DataDep}(b, c)}{\text{DataDep}(a, c)}
\] (4.9)

For brevity, we omit the full details of the flow-insensitive data-dependency analysis handling sequential program constructs (see, e.g., [96]), and instead illustrate its behavior through an example. Consider the statement \( a = x + y \), i.e., an add instruction. The semantics of this instruction can be modeled by generating the two facts: DataDep\((x, a)\) and DataDep\((y, a)\), i.e., \( a \) is dependent on both \( x \) and \( y \). Similarly, the statement \( x = z \), generates the fact DataDep\((z, x)\). Taking the transitive closure of DataDep deduces that DataDep\((z, a)\), i.e., \( a \) is dependent on \( z \).

We assume that each thread’s control-dependencies are in the relation ControlDep where ControlDep\((a, b)\) indicates that statement \( b \) is control-dependent on statement \( a \). This is efficiently computable using post-dominators [51].

Finally, the control- and data-dependency relations can be combined and have their transitive closure taken creating what is commonly referred to as the program dependency relation [51]. We define the program-dependency relation to be the relation Dep:
\[
\frac{\text{DataDep}(a, b) \land \text{ControlDep}(a, b)}{\text{Dep}(a, b) \land \text{Dep}(b, c)}
\] (4.10)

4.3.1 Proofs of Correctness

Finally, we state the soundness of the previous analysis.

**Theorem 3.** The weak-memory dependency analysis is a sound over-approximation of the dependencies in the program i.e., if there exists a dependency between two statements in the program then it will be included in the analysis.

**Proof.** As in prior work [96] a dependency analysis is sound if the dependencies in the transitive-reduction (non-transitively induced) for each statement are included. We model dependencies for fences and memory-barriers based on their semantics. Thus, the analysis contains all dependencies directly involving fences/barriers. We assume all dependencies involving non-fence/barriers are soundly considered. So, analysis includes the transitive-reduction of all dependencies. \( \square \)
4.4 Change–Impact Analysis under Weak Memory Models

Next, we integrate the memory barrier related dependencies into a change-impact analysis. We want to identify statements in the program either impacted by, or may impact, a modification. No existing analysis [29, 70, 131] handles weak-memory primitives.

We consider a program $P$ having statements added, removed, and/or modified to create $P'$. Identifying these statements can be accomplished using a diff tool (e.g., [134, 163]). $Add$, $Mod$ are the sets of added or modified statements in $P'$, and $Rem$ is the set of removed statements in $P$. Finally, $\Delta$ maps each unmodified statement in $P$ to the associated statement in $P'$. The goal is to compute $Impacted$ and $MayImpact$ containing statements in $P'$ either impacted by changes, or which may impact changed statements, respectively.

First, we introduce two functions computing the forward (Fwd) and backward (Bwd) slices of statements in a program [81, 162]. The forward-slice of a statement $s$ is the set of statements whose computation depends on $s$. The backward-slice is the set of statements on which $s$’s computation is dependent on. They are simply forward and backward reachability within the $Dep$ relation defined previously: $Fwd(s) = \{s' \mid Dep(s, s')\}$, and $Bwd(s) = \{s' \mid Dep(s', s)\}$.

ComputesImpacted, shown in Algorithm 8, returns $Impacted$, the set of impacted statements in $P'$. It operates as follows. First, $Impacted$ is initialized to the set of added or modified statements in $P'$ (line 6). Then, it is expanded to contain all items in the forward-slice in $P'$ of added/modified statements (line 7). Finally, we take the forward slice on each removed statement in $P$ and map the results to their counterparts in $P'$ (line 8). The forward slice is done in $P$ since removed statements do not exist in $P'$.

MayImpact is computed in ComputeMayImpact in the same way but uses the backward slice.

**Algorithm 8 Computation of $Impacted$ and $MayImpact$.**

1: $P$ and $P'$ are old and new program versions
2: $Add$ and $Mod$ are sets of statements added/modified in $P'$
3: $Rem$ is the set of statements removed in $P$
4: $\Delta$ maps statements in $P$ to their counterpart in $P'$
5: function ComputesImpacted
6: $Impacted \leftarrow Add \cup Mod$
7: $Impacted \cup \{s \mid s' \in (Add \cup Mod) \land s \in Fwd(s')\}$
8: $Impacted \cup \{s \mid s' \in Rem \land s'' \in Fwd(s') \land s \in \Delta(s'')\}$
9: return $Impacted$
10: function ComputeMayImpact
11: $MayImpact \leftarrow Add \cup Mod$
12: $MayImpact \cup \{s \mid s' \in (Add \cup Mod) \land s \in Bwd(s')\}$
13: $MayImpact \cup \{s \mid s' \in Rem \land s'' \in Bwd(s') \land s \in \Delta(s'')\}$
14: return $MayImpact$
4.5 Incremental Thread-Modular Abstract Interpretation

The incremental thread-modular abstract interpreter integrates the change-impact results to reduce runtime overhead. The reduction stems from two insights: first, we preserve memory states from the old version, and second, we identify statements whose transfer function can be ignored. This is sound since states unaffected by program modifications can be preserved; and we can ignore the transfer function of statements provably uninvolved with modified regions. This section discusses these points in detail.

Algorithm 9 Incremental analysis of thread $T$ in the presence of interference $I$ using prior analysis results $Prior$ and change impact information $Impacted$ and $MayImpact$

```plaintext
1: function IncAnalyzeThread($T = (N, n_0, \delta), I, Prior, Impacted, MayImpact$)
2: \hspace{1em} ▷ Initialize $S$, a map from nodes to states
3: $S(n_0) \leftarrow \top$, otherwise $S(n) \leftarrow \bot$
4: \hspace{1em} ▷ Preserve memory states from prior executions
5: for all $n \in N, n \notin Impacted$
6: $S(n) \leftarrow Prior(n)$
7: $W \leftarrow \emptyset$ ▷ $W$ is a set of nodes to process
8: for all $n \in Impacted$
9: $W \leftarrow W \cup \{n\}$
10: while $\exists n \in W$
11: $W \leftarrow W \setminus \{n\}$
12: if $n \notin Impacted \land n \notin MayImpact$
13: \hspace{1em} $s \leftarrow S(n)$ \hspace{1em} ▷ Use identity transfer-function
14: else if $n$ is a shared-memory read of variable $v$
15: \hspace{1em} $s \leftarrow tfunc(n, S(n) \cup I(v))$
16: else
17: \hspace{1em} $s \leftarrow tfunc(n, S(n))$
18: for all $(n, n') \in \delta$ such that $s \not\sqsubseteq S(n')$
19: $S(n') \leftarrow S(n') \cup s$
20: $W \leftarrow W \cup \{n'\}$
21: return $S$
```

First, we modify the analysis of a single thread (Algorithm 5) to use the change-impact analysis, as seen in Algorithm 9. The modifications are as follows: first, nodes which are not impacted have their memory states preserved from the prior analysis (lines 5–6). Second, the set of nodes to process, $W$, is initialized with any impacted node. This is required when preserving states from prior executions because simply starting the analysis at the entry node may cause changed regions of the program to be skipped. Consider a control-flow graph $n_0 \rightarrow n_1 \rightarrow n_2$ where $n_0$ is the entry node, $n_0$ and $n_1$ are not impacted, and $n_2$ is impacted. Since they are not impacted, the memory states for $n_0$ and $n_1$ are preserved. So, starting, as in Algorithm 5, with $n_0$ causes a fixed-point to be reached after processing $n_0$: the input to $n_1$ does not change from the prior version after executing $n_0$. As a result, $n_2$ is never executed even though it was impacted and may lead to different results relative to the prior analysis.

Additionally, nodes which are neither impacted nor may impact other nodes simply pass their input memory state as their output. This prevents the costly operation of executing a node’s transfer function. Furthermore, it is useful in practice because the change-impact
analysis analyzes the data-flow of the program whereas the abstract interpreter analyzes the control-flow: small portions of the data-flow of the program may be impacted separated by a large number of control-flow nodes causing a slowdown. Consider the following control-flow graph: \((n_0 : y = a) \rightarrow (n_1 : x = b) \rightarrow (n_2 : x++) \rightarrow (n_3 : assert(y \neq 5))\) Consider that operations involving \(a\) were impacted, i.e., nodes \(n_0\) and \(n_3\). Then, the computations involving \(b\) (nodes \(n_2\) and \(n_3\)) are immaterial. So, their transfer functions do not need to be executed, they can simply pass along the relevant information pertaining to \(a\) and \(y\).

Finally, Algorithm 10 incrementally analyzes a system of threads: it first performs the change-impact analysis and passes the result to the analyzer for individual threads (Algorithm 9).

**Algorithm 10** Analyze new version \(P'\) using the analysis results, \(Prior\) of old version \(P\)

1: function IncAnalyzeProg\((P, P', Prior)\)
2: \(\triangleright\) Impacted and MayImpact are sets of nodes which are impacted by, or may impact, other nodes, respectively
3: Impacted \(\leftarrow\) ComputeImpacted
4: MayImpact \(\leftarrow\) ComputeMayImpact
5: \(Ss \leftarrow\) map all nodes in \(P\) to \(\perp\)
6: \(Ss' \leftarrow Ss\)
7: repeat
8: \(Ss = Ss'\)
9: for all \(T \in P\)
10: \(I \leftarrow \bigcup\) Interf\((T', Ss)\) for each \(T' \in P, T' \neq T\)
11: \(Ss' \leftarrow Ss' \uplus\) IncAnalyzeThread\((T, I, Prior, Impacted, MayImpact)\)
12: until \(Ss = Ss'\)

Adapting the constraint-based abstract interpreter to be incremental follows similarly to Algorithms 9 and 10. Specifically, there are two modifications. First, on line 9 of Algorithm 7 a new function, IncAnalyzeThread-Mod, should be called which is the same as IncAnalyzeThread (Algorithm 9) except, as before, the second argument, the interference, pairs each load \(l\) to the interference it should read from. Second, any load \(l\) which is neither impacted nor may-impact does not need to have its interference combinations explored since the value loaded is immaterial. Thus, in InterferenceComboFeasible any such load can only be mapped to the singleton set \(\{s_{\text{dummy}}, e_{\text{self}}\}\) (lines 25–33).

### 4.5.1 Proofs of Correctness

We consider our analysis sound if when there exists an error in the program it is reported by the analysis. To do this, we show the two modifications to the thread-modular abstract interpreter, and the one modification to the constraint-based abstract interpreter are sound.

**Theorem 4.** Preserving the memory-states of statements unaffected by a change is sound.

**Proof.** Suppose not, i.e., preserving the memory-states of unaffected statements causes some error state to be reachable in a concrete execution of the program but is not reported as reachable by the analysis. Since it is the only modification to the abstract interpreter the reachability of this error state must have been caused by preserving the memory-state of an
unaffected statement, i.e., preserving memory-state \( s \) at some unaffected statement causes the error to be unreachable. So, \( s \) must be an unsound under-approximation of the program’s concrete behavior. Since the change-impact analysis is sound \( s \) must not be affected by any change in the program, so, the memory-state at \( s \) is the same in the current and prior version of the program. Since the thread-modular abstract interpreter is sound \( s \) is a sound over-approximation of the program’s concrete behavior. This contradicts our assumption.

**Theorem 5.** Using the identify transfer-function for statements which are neither impacted by nor may impact changed regions of the program is sound.

*Proof.* Suppose not, i.e., using the identity transfer-function for a statement which is either not impacted by or must not impact a changed region of the program causes some error state to be reachable in a concrete execution of the program but is not reported as reachable by the analysis. Since it is the only change, using the identify transfer-function for some statement \( s \) must cause the error-state to be unreachable. There are two cases: \( s \) is either not impacted by, or must not impact a changed region of the program.

For the first case, \( s \) is not impacted by some change. Thus, from Theorem 4 the memory state before and after \( s \) is a sound over-approximation regardless of if \( s \) is executed or not. So, the execution of \( s \) is immaterial to the reachability of an error.

For the second case, \( s \) must not impact the changed region of the program. Assuming the prior version of the program was safe then there are no reachable errors in the unchanged region of the program. So \( s \) cannot cause any new error state to be reachable.

In both cases, no new error state is reachable.

**Theorem 6.** Mapping a load \( l \) to an arbitrary interfering store/load (e.g., \((s_{\text{dummy}},e_{\text{self}})\)) within the constraint-based abstract interpreter is sound.

*Proof.* Follows from Theorem 5.

**Theorem 7.** The incremental thread-modular abstract interpreter is sound both with and without constraints.

*Proof.* Follows from Theorems 4, 5, and 6.

4.6 Experiments

We implemented the proposed method in a tool called InCA using the LLVM/Clang [12] compiler framework, and Apron [87] library for abstract domains. We evaluate against two non-incremental tools: first Min`e [119–121], the non-constraint thread-modular analyzer (Algorithm 6), and second FRUIT TREE [103], the constraint-based analyzer (Algorithm 7).
### Table 4.1: Summary of benchmarks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Impacted (%)</th>
<th>May Impact (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ib700wdt2-1</td>
<td>2.005</td>
<td>0.118</td>
</tr>
<tr>
<td>ib700wdt2-2</td>
<td>20.047</td>
<td>0.649</td>
</tr>
<tr>
<td>ib700wdt2-3</td>
<td>40.094</td>
<td>1.239</td>
</tr>
<tr>
<td>ib700wdt2-4</td>
<td>60.142</td>
<td>1.828</td>
</tr>
<tr>
<td>ib700wdt2-5</td>
<td>82.842</td>
<td>2.417</td>
</tr>
<tr>
<td>ib8xctco1</td>
<td>35.787</td>
<td>0.184</td>
</tr>
<tr>
<td>ib8xctco2</td>
<td>60.626</td>
<td>1.840</td>
</tr>
<tr>
<td>ib8xctco3</td>
<td>77.185</td>
<td>5.428</td>
</tr>
<tr>
<td>ib8xctco4</td>
<td>77.185</td>
<td>10.948</td>
</tr>
<tr>
<td>ib8xctco5</td>
<td>77.185</td>
<td>22.079</td>
</tr>
<tr>
<td>machzwd1</td>
<td>13.652</td>
<td>4.892</td>
</tr>
<tr>
<td>machzwd2</td>
<td>40.956</td>
<td>13.993</td>
</tr>
<tr>
<td>machzwd3</td>
<td>56.997</td>
<td>16.268</td>
</tr>
<tr>
<td>machzwd4</td>
<td>74.061</td>
<td>19.681</td>
</tr>
<tr>
<td>machzwd5</td>
<td>74.061</td>
<td>27.873</td>
</tr>
<tr>
<td>mixcomwd1</td>
<td>54.958</td>
<td>1.160</td>
</tr>
<tr>
<td>mixcomwd2</td>
<td>67.616</td>
<td>2.215</td>
</tr>
<tr>
<td>mixcomwd3</td>
<td>80.274</td>
<td>3.270</td>
</tr>
<tr>
<td>mixcomwd4</td>
<td>85.232</td>
<td>5.063</td>
</tr>
<tr>
<td>mixcomwd5</td>
<td>85.232</td>
<td>27.873</td>
</tr>
<tr>
<td>ib700wdt1</td>
<td>1.538</td>
<td>24.615</td>
</tr>
<tr>
<td>ib700wdt2</td>
<td>41.538</td>
<td>9.231</td>
</tr>
<tr>
<td>ib700wdt3</td>
<td>43.077</td>
<td>12.308</td>
</tr>
<tr>
<td>ib700wdt4</td>
<td>3.077</td>
<td>27.692</td>
</tr>
<tr>
<td>pcwd1</td>
<td>13.948</td>
<td>0.054</td>
</tr>
<tr>
<td>pcwd2</td>
<td>41.845</td>
<td>0.054</td>
</tr>
<tr>
<td>pcwd3</td>
<td>69.742</td>
<td>0.054</td>
</tr>
<tr>
<td>pcwd4</td>
<td>86.334</td>
<td>1.717</td>
</tr>
<tr>
<td>pcwd5</td>
<td>86.334</td>
<td>19.099</td>
</tr>
<tr>
<td>pcwdpc1</td>
<td>13.784</td>
<td>0.569</td>
</tr>
<tr>
<td>pcwdpc2</td>
<td>35.985</td>
<td>1.434</td>
</tr>
<tr>
<td>pcwdpc3</td>
<td>58.187</td>
<td>2.339</td>
</tr>
<tr>
<td>pcwdpc4</td>
<td>80.389</td>
<td>3.284</td>
</tr>
<tr>
<td>pcwdpc5</td>
<td>80.389</td>
<td>8.141</td>
</tr>
<tr>
<td>pcwdpc6</td>
<td>80.389</td>
<td>23.682</td>
</tr>
</tbody>
</table>

The experiments were performed on 15 distinct Linux device drivers [47, 102, 103] with a total of 74 different versions. The modified program versions were created by the authors by, e.g., changing arithmetic operators, or modifying branch conditions. We calculated the percentage of total statements in the modified programs either impacted by, or may-impact, the modification(s) to give a rough measure of how far reaching the modification was. The summary of the benchmarks are in Table 4.1. The name, percentage of impacted statements, and percentage of may-impact statements are shown in columns 1–3, respectively. Various modifications of the same program are separated by horizontal rules. Overall, the benchmarks are a diverse set of modifications with on average 57% of the statements impacted by a modification, ranging from 1.5% to 92%. In total, they span 72,838 lines of code.

Our experiments were designed to address the following:

- How effective is our incremental thread-modular analysis?
- Does our incremental analysis reduce the runtime of the constraint-based analyzer?
- Does our incremental analysis reduce the runtime of the non-constraint-based analyzer?
First, we compare the non-incremental constraint-based analyzer (\textit{FruitTree [103]}) and our constraint-based incremental approach in \textit{InCA}. We present only the results under TSO since the results for SC, PSO, and RMO were similar. Table 4.2 summarizes the results: column 1 shows the test name, the runtime/number of properties verified for \textit{FruitTree} and \textit{InCA} are shown in columns 2 and 3, and 4 and 5, respectively, and column 5 shows the reduction in runtime of our technique. The runtime of \textit{InCA} includes the time for the change-impact analysis. Overall, both methods finish on every test, thus verify the same number of properties, but the incremental approach of \textit{InCA} takes only 44 minutes whereas the non-incremental approach takes 4 hours. This is an 82% reduction in runtime.

Our approach finishes faster in all tests except mixcomwd5, and wdt977-5. Here, a significant portion of the program was modified so the overhead of performing and integrating the change-impact analysis outweighs its benefits. This is expected: our technique works best when modifications are small.
Next, we compare to the non-constraint based approach of Miné to the non-constraint based incremental approach in INCA. We did not use the monotonicity domain of [121]. This analysis is sound but not tailored for any memory model so there is only one set of results, summarized in Table 4.3.

Overall, non-constraint-based incremental analysis reduces runtime by 41%. This is less relative to the constraint-based reduction since using the change-impact to not consider a load within the interference combinations has exponential savings. Again, in some instances, e.g., pcwdpci6, our techniques is slower since large portions of the program are impacted.
4.7 Related Work

Incremental analyses were developed for symbolic execution for both sequential \[29,131,163\], and concurrent [70] programs. However symbolic execution considers the problem, orthogonal to property verification, of test-input generation. Additionally, the concurrent change-impact analysis [70] does not handle weak-memory primitives. Backes et al. [22] integrated a change-impact analysis to optimize the problem of functional equivalence checking again using symbolic execution. Similarly, they only considered sequential programs.

SymDiff [105] uses a symbolic encoding of two program versions and attempts to find properties violated in one version, while unviolated in another, i.e., indications of differences in behavior. They only handled sequential programs, and address a problem, finding trace witnesses for differences in program behavior, orthogonal to property verification via numerical abstract interpretation.

In regression testing, change-impact has enabled tests prioritization and thread-schedules generation after a program modification [85,153,168,169]. This notion of comparing concurrent executions to identify their semantic difference was introduced by Shasha and Snir [143] and extended by Bouajjani et al. [26], although in the latter case, bounded model checking was used. Like symbolic execution, these techniques are bounded and aim at bug finding rather than property verification.

Both static and dynamic [20,32,108,110,135,137] change-impact analyses have been widely studied. They focus on handling non-concurrency language features [20,21,109,135,137], or use dynamic information [21,108,130]. Here, we explicitly designed a static data-flow based change-impact analysis handling weak-memory primitives: this was not addressed by prior work.

Prior work [76] refined a data-flow based change-impact analysis, by checking for semantic equivalence induced by a change via a symbolic analysis. This increases the analysis’ accuracy by reducing the number of statements falsely labeled as modified. This is complementary to our work since any accuracy increases to the change-impact analysis will benefit our analysis by reducing the number of impacted/may-impact statements. It is also orthogonal: here, we focused on creating a weak-memory aware change-impact analysis and integrating it into a thread-modular analyzer rather than creating a new change-impact analysis.

We directly compared our work to the thread-modular abstract interpreters of Minè [119–121] and Kusano and Wang [102,103]. Both are non-incremental. Thread-modular techniques have also been applied in model-checking [56,59], including predicate abstraction [73,77]. These are also non-incremental and typically assume the existence of a finite-state model.

Various non-thread-modular verifiers for unbounded programs operate on data-flow graphs [47], use trace-sampling [49], or sequentialization [106] for weak-memory [39,100,117]. These are all non-incremental and orthogonal techniques to thread-modular abstract interpretation.
Additionally, a large body of work exists developing bug-hunting techniques for concurrent programs running under weak-memory such as bounded model checking [15–17], and stateless model-checking [58] for weak-memory [6, 9, 41, 82, 129, 171]. These techniques are all non-incremental, and bound the program’s execution length thus are not capable of verifying unbounded programs.

4.8 Discussion

We presented InCA, the first incremental thread-modular abstract interpreter. We developed a semi-flow-insensitive dependency analysis handling weak-memory primitives, formalized it into a change-impact analysis, and then integrated the change-impact analysis into both a constraint- and non-constraint based thread-modular abstract interpreter. Our experiments compared both approaches to their non-incremental counterparts and showed significant reductions in runtime overhead.
Chapter 5

Conclusions and Future Work

We presented the notion of a constraint-based thread-modular abstract interpreter, and showed how it naturally adapts to weak memory, and can be made incremental. Specifically:

In Chapter 2 we showed how a lightweight system of constraints can be used to prove the infeasibility of interferences considered during thread-modular abstract interpretation. This allows the analysis to be much more accurate compared to analyses considering all interference combinations as feasible. Furthermore, thanks to our optimizations, the analysis has only a moderate increase in runtime overhead compared to less accurate prior work.

In Chapter 3 we showed how the constraint-based analysis provides the natural framework to accurately analyze programs running under weak memory, particularly TSO, PSO, and RMO. We introduced a new constraint system capable of reasoning about interference feasibility assuming these memory models and showed it significantly outperforms, in terms of accuracy, all other thread-modular abstract interpreters capable of reasoning about weak memory.

Finally, in Chapter 4 we introduced a dependency analysis for weak memory, formulated it within a change-impact analysis, and then integrated it into the thread-modular abstract interpreter. This offers significant runtime reduction, relative to prior work, when considering the problem of incremental verification, i.e., analyzing a new program while assuming the old program has been verified previously.

5.1 Future Work

There are many directions for future work paved by the foundation of constraint-based thread-modular abstract interpretation presented in this dissertation.

First, there is still additional work to be done along the same line of research presented within this dissertation. For example, we evaluated the proposed techniques on mostly Linux device
drivers; this could potentially have biased the benefit of our techniques, i.e., our techniques may be less beneficial on different types of programs, e.g., client–server applications or databases. Thus, examining the empirical results on a more varied set of benchmarks would increase the confidence that this technique is widely applicable.

Expanding the tool to work on more benchmarks would inevitably come with new engineering problems to solve, for example, modeling intricate programming language features (e.g., vector operations) within the abstract interpreter, and ensuring the underlying alias analysis remains precise enough without high runtime. Along this direction, we would expect more exploration of abstractions: for example, for memory allocations (e.g., context-sensitivity), memory modeling (e.g., modeling the memory as bytes versus words, modeling arrays with dynamic lengths, specific handling of strings), and calling contexts. Overall, this would involve fine tuning the tool to be more scalable and reasonably accurate.

Similarly, we presented a set of deduction rules for pruning infeasible thread-interference combinations. These, again, were designed to reduce the number of false alarms within our benchmarks. It is likely that applying the tool to more programs would lead to more opportunities to create new deduction rules to further reduce the number of false alarms. Creating more of these deduction rules would require careful evaluation of their applicability (decreasing false alarms) and the cost (not incurring a high runtime overhead).

Since we presented a verification algorithm, our technique could be integrated into a syntax-guided program synthesis procedure \cite{19, 42–44, 170} to solve problems such as fence insertion, and the synthesis or repair of low-level synchronization primitives \cite{97}. Specifically, our verification algorithm could check if some automated syntactical change to a program, or an entire generated program, fulfills its specification. If not, the program synthesis loop would reject the candidate program and generate a new one. The iterative process continues until a suitable candidate is found.

The constraint system we presented targeted SC, TSO, PSO, and RMO. Additional weak-memory hardware models such as ARM and Power \cite{8,14}, as well as language-level memory models in Java and C/C++ \cite{99,114,129}, also exist. Furthermore, in this dissertation we focused on multithreaded programs with arbitrary preemption and no priorities. Different concurrency models, such as priority-based interrupts in microcontrollers \cite{72,151}, and event-driven programs in JavaScript/UI libraries are also concurrent \cite{33,150}, but are more restrictive than full multithreaded programs. Thus, directly applying our technique to these programs produces false alarms. It remains to be shown how to adapt our constraint system to handle these memory and concurrency models.

Another area not yet explored by this work is fault localization: our tool is only capable of proving that a program is correct, and, when reporting a bug, is unable to explain why it is a bug. For example, it would be useful to localize the relevant statements in the program causing the error, to either point the developer in the direction of the issue (if the reported bug is truly a bug), or provide hints as to why the false alarm was being generated (due to overapproximation). Such a technique, based on our experience, would greatly improve the
Our numerical abstract interpretation, like many analyses, depends on an accurate points-to analysis. Such a points-to analysis is performed before the more heavy-weight numerical analysis. This leads to inaccuracies since the points-to analysis is numerically oblivious: values such as array offsets are unknown during the points-to analysis and thus are assumed to be arbitrary. However, the points-to sets could be computed on-the-fly during the numerical analysis, thus allowing for information to be shared and accuracy to be improved overall.

Similarly, our model of variables within the numerical abstract domain in the program was coarse-grained. The thread-modular abstract interpretation paradigm is amenable to more fine-grained abstract domains such as those employed in a shape-analysis [138].

The constraint system used a Datalog engine for reasoning about Boolean variables with finite domains. The actual abstract interpretation, on the other hand, reasoned about numerical variables within infinite domains. The current presentation and implementation separated the two into disjoint components. More expressive solvers, such as Datalog modulo theory solvers, or prolog, could encapsulate the entire analysis. Such a monolithic encapsulation would permit the most sharing of information across components.

The constraint-based system largely analyzes scenarios involving two threads. There are some scenarios where reasoning about the feasibility of thread interferences may require assumptions based on three or more threads. Expanding the constraint system to handle them would allow for less false alarms.

We discussed a large number of thread-modular analysis techniques [24, 47, 49, 50, 56, 57, 69, 73–75, 77, 80, 84, 94, 113, 119–122, 139, 140]. Each approaches the problem differently, e.g., with various interference abstractions, and program representations. An empirical comparison between them, as well as exploring techniques to make use of these techniques simultaneously, would be beneficial.

Finally, our tool is capable of analyzing programs with infinite state. This puts it in a unique position of being able to work in environments where there are no restriction on either the number of threads executing (e.g., in concurrent data structures assuming no limitations on the client), or code running for infinite time (e.g., in an infinite loop). Exploring open problems in these domains requiring verification (e.g., proving the correctness of security related properties using non-numerical analyses for concurrent programs, such as taint analysis) is a possibility.
Chapter 6

Bibliography


