Extrapolation and Synthesis for Relaxed Memory Models

Yuri Meshman
Extrapolation and Synthesis for Relaxed Memory Models

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Yuri Meshman

Submitted to the Senate
of the Technion — Israel Institute of Technology
Cheshvan 5777    Haifa    November 2016
This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


### Acknowledgements

First and foremost, I would like to thank my supervisor, Eran Yahav, for his guidance and support, for pushing me, and for putting up with me. I would like to thank my parents, Elena and Michael, who are, for me, an eternal example of striving for excellence and being the best I could be. I would like to thank my close group of friends who made the process much more pleasurable. And, thanks Malish.

The Technion’s funding of this research is hereby acknowledged.
## Contents

### List of Figures

<table>
<thead>
<tr>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
</tr>
<tr>
<td>1.1 Motivating Example</td>
</tr>
<tr>
<td>2 Preliminaries</td>
</tr>
<tr>
<td>2.1 Relaxed Memory Models</td>
</tr>
<tr>
<td>2.1.1 Hardware Relaxed Memory Models</td>
</tr>
<tr>
<td>3 Predicate Abstraction for Relaxed Memory Models</td>
</tr>
<tr>
<td>3.1 Online Predicate Abstraction</td>
</tr>
<tr>
<td>3.1.1 Partial Coherence Abstraction for Relaxed Memory Models</td>
</tr>
<tr>
<td>3.2 Predicate Extrapolation</td>
</tr>
<tr>
<td>3.2.1 Boolean Program Based Predicate Abstraction</td>
</tr>
<tr>
<td>3.2.2 The Approach</td>
</tr>
<tr>
<td>3.2.3 Extrapolation Based Predicate Abstraction Overview</td>
</tr>
<tr>
<td>4 Numerical Abstraction for Relaxed Memory Models</td>
</tr>
<tr>
<td>4.1 Overview</td>
</tr>
<tr>
<td>4.1.1 Motivating Example</td>
</tr>
<tr>
<td>4.1.2 Searching for Fence Assignment and Refinement Placement</td>
</tr>
<tr>
<td>4.1.3 Refinement Placement - Reduction &amp; Abstraction</td>
</tr>
<tr>
<td>4.2 Abstraction-Guided Fence Synthesis</td>
</tr>
<tr>
<td>4.2.1 Abstraction-Guided Fence Synthesis</td>
</tr>
<tr>
<td>4.3 Automatic Verification</td>
</tr>
<tr>
<td>4.3.1 Reduction</td>
</tr>
<tr>
<td>4.3.2 Analysis with Numerical Abstract Domains</td>
</tr>
<tr>
<td>4.3.3 Abstraction Refinement of Numerical Analysis</td>
</tr>
<tr>
<td>4.3.4 Empty-Buffer Analysis</td>
</tr>
<tr>
<td>4.4 Evaluation</td>
</tr>
</tbody>
</table>

### Abstract

1. Introduction

1.1 Motivating Example

2 Preliminaries

2.1 Relaxed Memory Models

2.1.1 Hardware Relaxed Memory Models

3 Predicate Abstraction for Relaxed Memory Models

3.1 Online Predicate Abstraction

3.1.1 Partial Coherence Abstraction for Relaxed Memory Models

3.2 Predicate Extrapolation

3.2.1 Boolean Program Based Predicate Abstraction

3.2.2 The Approach

3.2.3 Extrapolation Based Predicate Abstraction Overview

4 Numerical Abstraction for Relaxed Memory Models

4.1 Overview

4.1.1 Motivating Example

4.1.2 Searching for Fence Assignment and Refinement Placement

4.1.3 Refinement Placement - Reduction & Abstraction

4.2 Abstraction-Guided Fence Synthesis

4.2.1 Abstraction-Guided Fence Synthesis

4.3 Automatic Verification

4.3.1 Reduction

4.3.2 Analysis with Numerical Abstract Domains

4.3.3 Abstraction Refinement of Numerical Analysis

4.3.4 Empty-Buffer Analysis

4.4 Evaluation
4.4.1 Concurrent Algorithms ........................................... 32
4.4.2 Results .......................................................... 32
4.5 Related Work ......................................................... 35

5 Effective Abstractions for Verification under Relaxed Memory Models ......................................................... 39
5.1 Overview .............................................................. 41
  5.1.1 Relaxed Behaviors .............................................. 41
  5.1.2 SC Equivalence vs. Flexible Safety Specifications .......... 41
  5.1.3 Our Approach .................................................. 42
5.2 Background .......................................................... 46
  5.2.1 Buffer Analysis ............................................... 46
  5.2.2 Direct Source-to-Source Encoding ......................... 47
  5.2.3 Direct TSO Translation ...................................... 48
  5.2.4 Direct PSO Translation ...................................... 50
  5.2.5 Shortcomings of the Direct Translation .................. 51
5.3 Abstraction-Guided Translation .................................. 51
  5.3.1 Robust Buffer Abstraction – Eliminating Buffer Shifting . 52
  5.3.2 Replacing Counters with Boolean Flags .................. 52
  5.3.3 New Translation Rules for TSO .......................... 53
  5.3.4 Soundness of the Robust Buffer Abstraction ............ 54
5.4 Evaluation ............................................................ 56
5.5 Related Work ........................................................ 60
5.6 Conclusion ........................................................... 61

6 Pattern-based Synthesis of Synchronization for the C++ Memory Model ......................................................... 63
6.1 Introduction .......................................................... 63
  6.1.1 The Problem .................................................. 64
  6.1.2 Our Approach: Pattern Based Synthesis of Synchronization . 64
6.2 Overview .............................................................. 64
  6.2.1 Running Example ............................................. 65
  6.2.2 Synthesizing Synchronization ............................... 66
6.3 C++ Relaxed Memory Model in a Nutshell ....................... 67
6.4 Synthesis of Synchronization ...................................... 69
  6.4.1 Atomic Memory Access Synchronization Synthesis .......... 69
  6.4.2 Patterns of Weak Memory Behavior ....................... 72
  6.4.3 Abstracting the Patterns .................................... 74
6.5 Implementation and Experimental Evaluation ................... 75
6.6 Related Work ........................................................ 77
7 Extrapolation

7.1 Model Checking with Symmetry ........................................... 79
7.1.1 Proof of Concept ....................................................... 80
7.2 Geometric Extrapolation ................................................... 85

8 Conclusions and Possible Future Directions Rising from This Work 87

8.1 Possible Directions Following our Predicate Abstraction Work .... 88
8.2 Possible Directions Following our Numerical Abstract Domains Analysis Work ................................................................. 90
8.2.1 Why Widening Hurts Verification? ....................................... 90
8.2.2 How to Do Analysis Debugging Using Widening Delay .......... 91
8.3 Possible Directions Following our C++ Relaxed Memory Model Synchronization Synthesis Work ................................. 91
8.3.1 Violation Patterns and Avoidance Templates are Not Complete . 91

Hebrew Abstract

i
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A simplified version of Lamport’s Bakery. (Initially Y1=Y2=0.)</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>PSO Relaxed Memory Model</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>k-buffer abstraction for PSO</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>An alternating bit protocol example with two threads</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Predicate abstraction: (a) classic algorithm, (b) with predicate extrapolation and cube extrapolation. Here, a rectangle represents an algorithm, while an oval represents input-output information (data)</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>A non-determinism example</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Peterson’s mutual exclusion algorithm</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Propagation of program correctness and abstraction refinements</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Translation of flag0=1 under PSO</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>PSO translation rules: each sequence is atomic</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Example program</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>The result of applying our transformation for TSO on Thread 1 from Fig. 5.1. The flush statements are not part of the program but need to be captured by the translation (and are inserted after every program statement)</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>The effect of a program trace on shared state and the state used by the two translations. The figure shows only statements of Thread 1 as well as flushes affecting Thread 1’s write buffer</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Direct TSO Translation Rules of TD</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Direct PSO Translation Rules of TD</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Abstraction-guided translation rules (i.e. TV) for TSO</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>Abstraction-guided translation rules (i.e. TV) for PSO</td>
<td>53</td>
</tr>
<tr>
<td>5.8</td>
<td>Translation domains</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>Dekker’s mutual exclusion algorithm. Variables flag[0], flag[1] and turn are declared as atomic locations and initialized to 0. The subscripts indicate the synchronization (consistency) annotations synthesized by our tool</td>
<td>65</td>
</tr>
</tbody>
</table>
6.2 An error trace containing two violation patterns: (a) store buffering (SB) and (b) load buffering (LB). These patterns were detected by our tool when analyzing Dekker’s algorithm.

6.3 Patterns of behaviors possible under C++ RMM but not under SC. Every column depicts the actions of one thread. We denote by store $x, 1$ a write of value 1 to location $x$ and by load $x(1)$ a read of value 1 from $x$. We assume that the initial value of $x$ and $y$ is 0.

6.4 Abstract patterns of behaviors possible under C++ RMM but not under SC.

7.1 CUDA array summation example.
7.2 CUDA example predicates
7.3 CUDA example proof for tid0=0
7.4 CUDA example mirrored predicates
7.5 CUDA example mirrored proof for tid1=1
7.6 Checking if one square subsumes the other
7.7 Rotated setting – checking if one square subsumes the other

8.1 Extrapolation with Interpolation
8.2 Visualization of the fr relation (ws orders all writes to a single memory location.)
8.3 A simplified version of Lamport’s Bakery. (Initialy $Y1=Y2=0$)
Abstract

In modern architectures memory operations may be reordered and executed non-atomically. Consistency is guaranteed for single thread execution, but not across threads, making it hard to reason about program correctness. Both software and hardware Relaxed Memory Models are defined to capture the possible execution behaviors where consistency is not guaranteed. Each Relaxed Memory Model poses its own challenges to successful program verification, and, in cases where verification fails, poses its own challenges to automatic synthesis of program corrections.

Our work covers Intel’s x-86 TSO and PSO hardware buffered memory models, as well as the C++RMM software memory model. We explore such abstraction techniques as Predicate Abstraction and Numerical Abstract Domains, to deal with both finite-state and infinite-state domains. In addition, we explore techniques to automatically synthesize program corrections when verification fails. The program corrections, are either memory fences or memory order parameters for memory operations, in accordance with the Relaxed Memory Model.

Where possible, we utilize the proof of program correctness from the more intuitive SC domain, extrapolating it to alleviate the complexity of proving program correctness under the Relaxed Memory Model. In addition, we show how to apply the approach of proof extrapolation from a simple to a more complex domain in domains other than Relaxed Memory Model.

We show that our techniques can synthesize fences in challenging concurrent algorithms such as the Ticket mutual exclusion algorithm, and several versions of Work Stealing Queues, and that they can synthesize, memory order parameters for algorithms such as RCU.
Chapter 1

Introduction

The challenge that we address in this work is twofold: (i) how to perform verification of a program \( P \) under some challenging, Relaxed Memory Model operational semantics, \( M \); and, (ii) if \( P \) violates its correctness criteria, how to synthesize a new verifiable program \( P' \), while preserving as many behaviors as possible of the original program \( P \).

The memory models that served as our Relaxed Memory Model, \( M \) are: In chapters 2-6, three generally known memory models Partial Store Order (PSO), Total Store Order (TSO), and C++ Relaxed Memory Order (C++RMM); And, in Chapter 7 two additional, ad-hoc domains. We will describe PSO and TSO in Chapter 2, C++RMM in Chapter 6, and the ad-hoc domains in Chapter 7. But, before entering into details about Relaxed Memory Models, we begin by showing a motivating example.

1.1 Motivating Example

In Fig. 1.1 we present a simplified version of Lamport’s Bakery algorithm [Lam74], which exemplifies the type of algorithms we attempt to verify and correct throughout this work. We will use it to show some challenges in reasoning about program executions under a Relaxed Memory Model.

In this algorithm, there are two executing processes, each trying to enter its critical section (line 4) while the other process is outside its critical section. Two global counters, \( Y_1 \) and \( Y_2 \), are used by the two processes for communication and synchronization. The

\begin{verbatim}
T1
while (true)
1: Y1 = Y2 + 1;
2: wait (PC_1 \neq 1);
3: wait (Y_2 = 0 \lor Y_1 \leq Y_2);
4: nop; //critical section
5: Y1 = 0;
end

T2
while (true)
1: Y2 = Y1 + 1;
2: wait (PC_2 \neq 1);
3: wait (Y_1 = 0 \lor Y_1 > Y_2);
4: nop; //critical section
5: Y2 = 0;
end

Assert(PC_1 \neq 4 \lor PC_2 \neq 4)
\end{verbatim}
counters holds for each process a number it chose based on the state of the other process. These counters are initially zero.

An example of Bakery execution: Process T1 reaches line 1 while observing the value of Y2 to be zero and sets its number (the value of Y1) to be one. Process T2 starts its execution, observes the value of Y1 to be one, sets its number (the value of Y2) to be two, and proceeds to line 2. Process T1 resumes its execution, observes T2 is after line 1 (is not choosing a number), then observes that its number is lower than Y2, and enters the critical section. Now, since Y1 is both not zero and is smaller than Y2, T2 can’t proceed past line 3 and so it waits. This is important to preserve mutual exclusion. Process T1 then proceeds to exit the critical section, sets the value of Y1 to zero, and proceeds to the next iteration.

Standard Bakery verification goal We would like to ensure Safety — Mutual Exclusion — for bakery: We would like to automatically ensure that both processes cannot reach the critical section at the same time.

For the given version of Bakery, the aforementioned verification goal holds (as was manually proven in [Lam74]) and can be checked with existing automated techniques. For example, to verify this program using predicate abstraction, we can choose predicates such as \{Y_1 \leq Y_2, Y_1 = 0, Y_2 = 0, \ldots\}, and use an existing predicate abstraction tool. This way we will automatically establish the correctness of this program under the standard interleaving execution scheme, the non-Relaxed Memory Model semantics, Sequential Consistency (SC).

A more relaxed semantics? However, imagine that after verifying Bakery under SC, we would like to consider its correctness under a more relaxed semantics, for example one that does not store a value to memory immediately on a write, but instead uses non-atomic stores. This means that a stored value is first written to local buffer, unseen by the other running process, and later flushed at some arbitrary time to main memory, only after-which the other process can observe it.

An immediate question is whether this program is even correct under a relaxed model of this type? The answer in this case is negative, as the following example shows.

Bakery TSO error trace Process T1 reaches line 1 while observing the value of Y2 to be zero and sets the value of Y1 to be one. The value is buffered, so locally T1 observes the value of Y1 to be one while globally the value (as observed by T2) is still zero! Process T2 starts its execution, observes the value of Y1 to be zero, sets the value of Y2 to be one globally, i.e. the value is written locally and immediately flushed to main memory, and proceeds to line 2. Process T1 resumes its execution, observes T2 is after line 1 (is not choosing a value for Y1), then observes that its number is equal to Y2 (both are one) and enters the critical section. Process T2 resumes its execution, observes that T1 is after line 1 (is not choosing a a value for Y2), then observes that
Y1 is still set to zero and enters the critical section. T1 and T2 are at their respective critical sections at the same time — a mutual exclusion breach.

To guarantee correctness (preventing all error traces), under the described relaxed semantics, the program would require the use of synchronization, in this case the adding of instructions called memory fences. We can prevent the above mentioned error trace by putting a memory fence after the write instruction at line 1. Execution of the fence will force all buffered values to be flashed to main memory, thereby making the last value written by T1 to be visible to T2, which in turn, will prevent the above-mentioned error trace. To prevent all possible error traces additional memory fences may be required.

**Goals**  
Our goals are: 1. to automatically verify algorithms such as Bakery under Relaxed Memory Models, and 2. if the program doesn’t satisfy its specification under the Relaxed Memory Model, to automatically generate a new program with added synchronization to restore correctness.

**Contributions**  
Our contributions are as follows:


We show that our techniques can synthesize fences in challenging concurrent algorithms such as the Ticket mutual exclusion algorithm, and several versions of Work Stealing Queues, and that they can synthesize memory order parameters for algorithms such as RCU. We also provide practical tools that perform these tasks.
Chapter 2

Preliminaries

In this section we introduce the various memory models referenced in later sections. We
will elaborate more on hardware memory models since they are discussed throughout
most of the chapters. The software memory model will be further elaborated on in
Chapter 6.

2.1 Relaxed Memory Models

Relaxed Memory Models are prolific in computer science fields but, for the most
part, remain hidden due to their intricate behavior. All modern concurrent hardware
implements behaviors that can be described by Relaxed Memory Models. They do so
to reduce execution time by utilizing the power of concurrency. Higher order languages,
usually, hide the hardware relaxed behaviors, and by default provide the user instructions
with over-restrictive synchronization built in. As a result, most programmers are not
only unaware of the possible erroneous behavior of their program if it was executed as
written, but they also miss opportunities for faster execution. To make use of the benefits
that relaxed behaviors offer, the programmer must resort to lower level languages or
make explicit use of special structures provided by the language for this purpose.

**Hardware Relaxed Memory Models**  By using a lower level language, the pro-
grammer benefits from greater control over the program execution for the specific
hardware and has to deal with a single hardware Relaxed Memory Model specific to
that hardware. (We note that our discussion here pertains only to one aspect of the
hardware captured by the Relaxed Memory Model; other aspects, such as cache, are
excluded.)

**Software Relaxed Memory Models**  When the programmer uses lower level lan-
guages, the specificity of the hardware is also a drawback since the code will need to
undergo revisions when it is ported to a different machine. Software Relaxed Memory
Models and their high-level language structures can be used to mitigate this drawback,
as software Relaxed Memory Models provide cross-platform compatibility; at the same
time, they benefit from the convenience of a higher order language, as well as from the
relaxed memory optimization. However, because these models have taken into account
any underlying hardware Relaxed Memory Model, they must also take into account the
most relaxed possible setting (allowing the most relaxed behaviors) and, in some cases,
behaviors even more relaxed than any of the memory models. (We will see an example
of this in the case of the C++RMM in Chapter 6.)

**Sequential Consistency**

The first memory model we present, *Sequential Consistency*, is also the most intuitive. Given a program of 2 (or more) processes, under Sequential Consistency, each execution of the program is consistent with an execution in which an interleaving of the processes instructions was chosen, followed by sequential execution
of the instructions according to the chosen interleaving. This is the memory model assumed in most cases when reasoning about concurrent program execution. We will use it as a baseline in future discussions, and all other Relaxed Memory Models will be relaxations of it. That is, the set of all possible behaviors under Sequential Consistency will be contained in the set of program behaviors under any of the Relaxed Memory Models we will discuss next.

We will begin by reviewing Hardware Relaxed Memory Models, specifically, PSO and x-86 TSO. Then we will briefly discuss the Relaxed Memory Models of Power and ARM processors.

### 2.1.1 Hardware Relaxed Memory Models

**Out-of-Order Execution**

For intuition about the necessity of relaxed memory for hardware we will first discuss *Out of Order Execution*. Modern processors greatly benefit from reordering instructions. For example, to mitigate writing to memory latency, they will postpone store instruction and execute them together (especially if they are stores to adjacent locations). For a single executing process, the processor maintains structures to provide coherence guarantees, so the process will not observe that its instructions are executed out of order. But, those guarantees are relaxed for multiple processes and one process can see the instructions of the other out of order.

**Execution Models — Store Buffer Based Memory Models**

To model behaviors of concurrent execution such as the one stemming from out of order execution many memory models use a fifo buffer. We will present two such memory models, TSO and PSO, but other buffered memory models exist as well (e.g., *Relaxed Memory Order*, RMO).

**PSO**

The *Partial Store Order* (PSO) memory model is shown in Fig. 2.1 for processes P1 and P2. As the name suggests, partial order between stores is maintained under
this memory model. That means that while the model allows reordering of instructions for one process, it guarantees that all other processes will observe that process’s stores to the same location in the same order they were committed for execution (although stores to different locations might be observed out of order). The model likewise guarantees the above by assigning to each process a buffer for each shared location (depicted in Fig. 2.1 via a cell array).

We will next describe the various memory operations in the model. The store instruction stores the value to the top of the buffer (the leftmost element of the buffer in Fig. 2.1). The load instruction attempts first to read the top value of the relevant variable (again, the leftmost element of the buffer in Fig. 2.1). This insures that each process will read the last value it has written. If the buffer is empty, the value is loaded from main memory. The model also contains a flush operation which can be injected and executed, by the system, at nondeterministic times during the execution. This operation removes the value from the bottom of the buffer (now the rightmost element of the buffer in Fig. 2.1), which is the oldest value in the buffer, and writes it to main memory. When implementing this operation it is common practice to include the shifting of values following the flush. This shifting can incur significant overhead for verification and is discussed in Chapter 5.

Finally, the model provides the programmer with a fence instruction she can use throughout the code. Following the execution of the operation, the buffer will become empty and the last written value to the buffer is written to main memory. Therefore, all...
other processes observe the last values written by the executing process before execution of the fence. There are various types of fence operations. In this work we discuss fences after the execution of which all the buffers of the executing process are empty (as opposed to, for example, location-specific fences). One drawback of fence operations, is the significant execution overhead they incur. One implementation will be a blocking wait, waiting until flush operations that clear the buffer are executed. Therefore, while fences are beneficial for algorithm correctness (for example, to prevent the Bakery TSO error trace in Section 1.1), they should be employed minimally if we still wish to benefit from relaxed memory optimized executions.

x86- TSO The Total Store Order (TSO) memory model is widely used in Intel processors and is very similar to the PSO memory model, except that for each process, it guarantees that all other processes in the system will observe all its stores in the same order in which they were committed for execution by that process. (The stores can still be observed with a delay, as is the case in Section 1.1.) The model guarantees the above by assigning to each process a single buffer for all shared location. (The buffer is similar to the depiction for PSO in Fig. 2.1, via a cell array, only that it will be a single array per process and not three.) While in PSO the buffer contained only shared variable values, in TSO each buffer entry is a pair consisting of a shared variable and value. Nonetheless, for all memory instructions, the buffer entries in both models are managed similarly.

Power To conclude this chapter, we will briefly discuss one relaxed memory execution model that is not store-buffer based. The Power memory model, developed by Sarkar et al. [SSA+11] and nicely explained in [Bat14], captures behaviors of common Power and ARM architectures and is more relaxed than the two previously mentioned models. One behavior unique to this model is that the loads of processes may be committed out of order. For example, in Fig. 1.1, at line 3, each of the two processes can read the value the other process will write at line 5, both can enter the critical section at the same time, and both can arrive at line 5, each one writing the value the other has previously read. (This behavior, called load buffering, is discussed further in Chapter 6.) The memory model description contains a storage subsystem to which processes make read and write memory access requests, and act according to the subsystem’s response. We will not deal with this memory model directly in the next chapters but it will be mentioned in the context of C++RMM.
Chapter 3

Predicate Abstraction for Relaxed Memory Models

In this chapter we shall review various approaches to predicate abstraction under relaxed memory models explored in recent years. We will discuss chiefly the work of Andrei et. al. [DMVY13], which will be referred to again in Chapter 7.

3.1 Online Predicate Abstraction

We begin with a description of a direct application of predicate abstraction to a specific Relaxed Memory Model by our colleagues Abdulla et al. [AAC\textsuperscript{+}12], developed at the same time as [DMVY13] and published before it. The work of [AAC\textsuperscript{+}12] builds on [KVY11] and handles infinite state programs (on Total Store Order) by applying both predicate abstraction and store buffer abstraction. We begin by describing the abstraction of Kuperstein et al., after which a short description of Abdulla et al.’s, predicate abstraction for Relaxed Memory Model, will follow.

3.1.1 Partial Coherence Abstraction for Relaxed Memory Models

One of the main challenges of Buffer Based Relaxed Memory Models is that the buffer can grow to an arbitrary size. This will make verification difficult for programs with a large buffer size, and impossible for programs without a buffer size bound.

Kuperstein et at. [KVY11] observe that in most programs, retaining full precision of the buffer structure is not required for sound verification and can be abstracted away. Their approach consists of storing most of the buffer in an unordered set, while retaining sufficient additional information about the buffer to maintain verification soundness. The abstraction will be described for PSO, but it is applied similarly for any other buffer based Relaxed Memory Model.

Before describing the abstraction, we note that concrete PSO semantics preserve the following 3 properties:
1. Intra-process coherence: If a process stores several values to shared variable $x$ and then performs a load from $x$, it should not see any value it has itself stored except the most recent one.

2. Inter-process coherence: A process $p_i$ should not observe values written to shared variable $x$ by process $p_j$ in an order different from the order in which they were written.

3. Fence semantics: If a process $p_i$ executes a fence when its buffer for variable $x$ is non-empty, the value of $x$ visible to other processes immediately after the fence should be the most recent value $p_i$ wrote.

These 3 properties guide towards the information that any abstraction of the buffer must retain. Kuperstein's abstraction is parametrized by a value $k$ that dictates the buffer abstraction precision. The k-buffer abstraction is described in Fig. 3.1 and, deriving from the previously mentioned properties, consists of: (i) the most recent value written to the buffer, (ii) $k$ of the least recent values written to the buffer, stored in-order, and (iii) the rest of the intermediate buffer location values, stored in an unordered set.

Intra-process coherence is maintained by storing the most recently written value to the buffer. Inter-process coherence is maintained first by the ordered $k$-sized portion of the buffer, and second by going over all possible retrieval orders of values from the unordered set portion of the buffer.

Kuperstein et al. also described a novel approach for fence inference, omitted here. The described abstraction was applied for verification and fence inference of several finite state mutual exclusion and synchronization algorithms. It was shown to be successful for $k=0$ for some algorithms (e.g. Peterson and Dekker) but not sufficiently precise for
others (e.g. a version of lamports fast mutex) where larger k sizes were used. For more
details we refer the reader to [KYZY11] and now proceed to describe the work of Abdulla
et al.

3.1.2 Automatic Fence Insertion in Integer Programs

The input to the algorithm of Abdulla et al. [AAC12] is: (i) a concurrent program
$Prog$, (ii) a value $k$, (iii) an initial set of predicates $Pred$, and (iv) a function $Bad$, over
the predicates, to mark error states.

Their approach consists of several reasoning stages: They begin by constructing from
$Prog$ a TSO program transition system. Because this transition system is potentially
unbounded, direct reachability analysis on it might not terminate (due to variable
values and buffer size). It is used, however, to check spuriousness of counterexamples.
From the TSO program transitions system and the input $k$, a $k$-TSO style abstraction
[KYZY11](shown in Section 3.1.1), is constructed where the first $k$ stores to the buffer
each thread are stored in order, the last store is saved, and the rest of the buffer is
stored in an unordered set. Finally, $Pred$, the initial predicate set, is used to construct
a transitions system $(f, pc, B_x, S)$ where: $f$ is the propagated combination of predicates
(initially true), $pc$ is the program counter, $B_x$ are the buffer location positions (the
values will be captured by predicates), and $S$ are the variables in the buffer with index
higher then $k$. The initial states are chosen such that they satisfy $Pred$.

The latter transition system is used to search for property violation. When an error
trace is found, it is propagated back through the aforementioned reasoning stages to
determine its type and the proper response to it, as will be explained next.

They perform CEGAR analysis, where they distinguish 4 types of possible coun-
terexamples(i) feasible under SC, (ii) feasible under TSO (but not SC), (iii) feasible
under k-TSO (but not TSO), and (iv) feasible under the given predicate set (but not
k-TSO). In case (i), the program is unsafe, and an appropriate message can be returned
to the user. In case (ii), they add memory fences and repeat the verification attempt.
In case (iii), the value of $k$ can be increased (to a finite value since the error trace is
finite). Finally, for case (iv), they refine the predicates. If no error trace is found, the
program is verified.

This approach has its advantages but it is specific to a single RMM and requires
building a predicate abstraction verification framework from scratch for each Relaxed
Memory Model, a task that can prove to be difficult. A more modular approach was
taken by Dan et al. [DMVY13]. The approach there was to utilize existing predicate
abstraction frameworks for Sequential Consistency and, modularly and soundly, lift
them for Relaxed Memory Model verification.
3.2 Predicate Extrapolation

We begin with a declaration of the verification problem generally for any Relaxed Memory Model M:

**The problem** Given a program $P$, a specification $S$ and a memory model $M$, we would like to answer whether $P$ satisfies $S$ under $M$, denoted as $P \models_M S$.

The verification abstraction scheme will be *Predicate Abstraction* based on *Boolean Programs*, so before we proceed we will spend a little time explaining what *Boolean Program Based Predicate Abstraction* is.

3.2.1 Boolean Program Based Predicate Abstraction

Given a program $P$ and a vocabulary (set of predicates) $V = \{p_1, \ldots, p_n\}$ with corresponding boolean variables $\hat{V} = \{b_1, \ldots, b_n\}$, standard predicate abstraction (e.g. [GS97, BMMR01]) constructs a boolean program $BP(P, V)$ that conservatively represents the behaviors of $P$ using only boolean variables from $\hat{V}$. When considering predicate abstraction in the context of relaxed Memory Models, two key challenges need to be addressed: (i) *soundness*: the boolean program must faithfully abstract the behaviors of $P$ running on model $M$; (ii) *predicate discovery*: there should be a mechanism for automatically discovering predicates that enable successful verification of $P$ running on memory model $M$.

**Soundness** Under sequential consistency (SC), predicate abstraction is sound and we know that $BP(P, V) \models_{SC} S$ implies $P \models_{SC} S$. Unfortunately, we observed that this does not necessarily hold in a direct approach to predicate abstraction for Relaxed Memory Models (see [DMVY13]).

**Predicate discovery** A key challenge with predicate abstraction is to discover a set of predicates that enable verification. Following classic abstraction refinement, one would start with a program that is to be verified on a particular relaxed model together with an initial set of predicates. Then, one would proceed to iteratively apply refinement until a set of predicates is found under which the program verifies (or the process times out). This might take a lot of time for Relaxed Memory Models.

We take a different approach to predicate discovery for programs running on RMMs. In our approach, we first obtain the predicates that enable verification of the program on sequential consistency (SC). Then, we automatically *extrapolate* from these SC predicates to produce a new set of predicates that can be used as a basis for verification on the relaxed model.
3.2.2 The Approach

Given a program $P$, a specification $S$ and a memory model $M$, our approach consists of the following steps:

1. **Verify under SC**: find a set of predicates $V$, sufficient to verify $P$ under sequential consistency, i.e., a set $V$ such that $\mathcal{BP}(P, V) \models_{SC} S$.

2. **Reduce to SC**: automatically construct a new program $P_{M}$ such that if $P_{M} \models_{SC} S$ then $P \models_{M} S$. The program $P_{M}$ contains an abstraction of the store buffers used in $M$.

3. **Discover new predicates**: automatically compute a new set of predicates $V_{M}$ that are used for predicate abstraction of $P_{M}$. This is a challenging step and the key idea is to leverage the verification of $P$ under SC.

4. **Construct a new boolean program**: given the new program $P_{M}$ and the new predicates $V_{M}$, automatically construct a boolean program $\mathcal{BP}(P_{M}, V_{M})$ such that $\mathcal{BP}(P_{M}, V_{M}) \models_{SC} S$ ensures that $P_{M} \models_{SC} S$, which in turn guarantees that $P \models_{M} S$.

5. **Check**: whether $\mathcal{BP}(P_{M}, V_{M}) \models_{SC} S$.

**Main Contributions**

- We provide a novel approach for predicate abstraction of programs running on Relaxed Memory Models, extrapolating from the predicate abstraction proof of the same program for sequential consistency.

- One of our insights is that the predicates used to verify $P$ under SC can be automatically extrapolated to discover new predicates for verification of the program with $M$ encoded in it, $P_{M}$.

- We instantiated our approach for the x86-TSO and PSO memory models. We implemented our approach and applied it to verify several challenging concurrent algorithms (both finite and infinite-state) under these models. We show that extrapolation is powerful enough to verify these algorithms.

3.2.3 Extrapolation Based Predicate Abstraction Overview

In this section, we give an informal overview of our approach using simple examples.

**Motivating Example**

Fig. 3.2 shows an implementation of an infinite state alternating bit protocol (ABP) with two concurrent threads. We use capitalized variable names to denote global shared variables, and variable names in lowercase to denote local variables. In this program,
global shared variables \( \text{Msg} \) and \( \text{Ack} \) have an initial value 0. We use this algorithm as our illustrative example; additional examples are discussed in [DMVY13].

**Specification** When executing on a sequentially consistent memory model, this program satisfies the invariant:

\[
((lRCnt = lSCnt) \lor ((lRCnt + 1) = lSCnt)).
\]

Here, the local variable \( lRCnt \) is the local counter for the receiver thread containing the number of received messages. Similarly, local variable \( lSCnt \) is the local counter for the sender thread containing the number of sent messages.

**Predicate Abstraction under Sequential Consistency** A traditional approach to predicate abstraction is shown in Fig. 3.3(a). To verify that ABP satisfies its specification under SC, we instantiate predicate abstraction with the following predicates:

\[
(Msg = 0), (Ack = 0), (lSSSt = 0), (lAck = 0)
\]

\[
(lMsg = 0), (lRSt = 0), (lRCnt = lSCnt), ((lRCnt + 1) = lSCnt)
\]

The result of predicate abstraction using these predicates is a concurrent boolean program that conservatively represents all behaviors of the original ABP program. In Fig. 3.3(a), this boolean program is denoted as the oval labeled Boolean Program B.

In the worst case, the construction of the concurrent boolean program, following standard predicate abstraction techniques (e.g., [GS97, BMMR01, FQ02]), involves an exponential number of calls to an underlying theorem prover. A critical part of the construction of the boolean program is searching the “cubes” — conjunctions of predicates — that imply a certain condition. This search is exponential in the number of predicates.

**An informal overview of PSO** In the partial-store-order (PSO) memory model, each thread maintains a private buffer (a sequence) for each global variable. When the thread writes to a global variable, the value is enqueued into the buffer for that variable. Non-deterministically, the values can be dequeued from the buffer and written to main memory. When the thread reads from a global variable, it first checks whether the buffer is empty and if so, it reads as usual from main memory. Otherwise, it reads the last value written in the buffer. The model is further equipped with a special *fence*
The challenges we address are: (i) how to verify programs such as ABP under Relaxed Memory Models such as x86 TSO and PSO, and (ii) how to deal with the exponential complexity of standard predicate abstraction in our RMM setting.
Predicate Abstraction for Relaxed Memory Models

In Fig. 3.3(b), we illustrate the ingredients and flow of our approach for solving the verification problem under Relaxed Memory Models. The figure illustrates two approaches for adapting predicate abstraction to our setting: Predicate Extrapolation and Cube Extrapolation (which includes Predicate Extrapolation, and is explained in [DMVY13]). Next, we discuss the steps of our approach.

**Step 1: Verify \( P \) under SC** The first step is to verify the program \( P \) under sequential consistency using standard predicate abstraction as outlined earlier in Fig. 3.3(a). Once the program is verified, we can leverage its set of predicates as well as its boolean program in the following steps.

**Step 2: Construct the reduced program \( P_M \)** This step is called “Reduction” in Fig. 3.3(b). To enable sound predicate abstraction of a program \( P \) under a relaxed memory model \( M \), we first reduce the problem into predicate abstraction of a sequentially consistent program. We do so by constructing a program \( P_M \) that conservatively represents the memory model \( M \) effects as part of the program.

The key idea in constructing \( P_M \) is to represent an abstraction of the store buffers of \( M \) as additional variables in \( P_M \). Since the constructed program \( P_M \) represents (an abstraction of) the details of the underlying memory model, we can soundly apply predicate abstraction to \( P_M \). The formal details of the reduction for x86-TSO and PSO are discussed in [DMVY13]. Here, we give an informal description.

For PSO, it is sufficient to consider a program \( P_{PSO} \) where every global variable \( X \) in \( P \) is also associated with: (i) additional \( k \) local variables for each thread \( t \): \( x_{1,t}, \ldots, x_{k,t} \), representing the content of a local store buffer for this variable in each thread \( t \), (ii) a buffer counter variable \( x_{cnt,t} \) that records the current position in the store buffer of \( X \) in thread \( t \).

The x86 TSO model maintains a single local buffer per process. This buffer is updated with stores to *any* of the global variables. However, we need additional variables to capture information about which global variable is stored in the buffer. The \( lhs \) variables contain the index of which global variable is addressed for each buffer element. The other variables are similar to PSO: (i) \( k \) local variables for each thread \( t \): \( lhs_{1,t}, \ldots, lhs_{k,t} \), representing the index of the global variable stored at a local store buffer in each thread \( t \), (ii) \( k \) local variables for each thread \( t \): \( rhs_{1,t}, \ldots, rhs_{k,t} \), representing the value content of a local store buffer in each thread \( t \), (iii) a buffer counter variable \( cnt,t \) that records the current position in the store buffer of thread \( t \).

**Step 3: Discover new predicates for \( P_M \)** After verifying \( P \) under SC and constructing \( P_M \), the remaining challenge is to discover a sufficient set of predicates for verifying that \( P_M \) satisfies a given specification. One of our main insights is that for buffered memory models such as x86 TSO and PSO, the predicates (and boolean program) used for verifying \( P \) under SC can be automatically leveraged to enable verification of \( P_M \). This step corresponds to the “Extrapolate” box. This step takes as input the set of
predicates $V$ that were successful for verifying $P$ under SC and outputs a new set $V_M$.

**Predicates for the motivating example** Next, we illustrate via our running example how to generate new predicates under PSO (the process under x86 TSO is similar).

Consider again the ABP algorithm of Fig. 3.2 and the 8 predicates listed earlier in Section 3.2.3. Following the structure of the additional variables in $P_M$, we can introduce additional predicates by cloning each predicate over a global variable $X$ into new predicates over store-buffer variables $x_1, \ldots, x_k$. For example, assuming $k = 1$, in addition to $Msg = 0$, we introduce $Msg_1 \cdot 0 = 0$.

To keep track of the buffer size for each buffered variable, we introduce additional predicates. For instance, for a global variable $Msg$, to register possible values for $Msg_{\cdot cnt} \cdot 0$, assuming $k = 1$, we introduce $Msg_{\cdot cnt} \cdot 0 = 0$ and $Msg_{\cdot cnt} \cdot 0 = 1$. Note that we do not introduce predicates such as $Msg_{\cdot cnt} \cdot 1 = 0$ and $Msg_{\cdot 1} \cdot 1 = 0$, as thread 1 always accesses $Msg$ via main memory. Another predicate we could have added is $Msg_{\cdot 1} \cdot 0 = Msg$. This predicate is not needed for the verification of ABP; however, we observe that such predicates can greatly reduce the state space of the model checker.

[DMVY13] discusses rules and optimizations for generating new predicates for the PSO memory model and presents the details for the x86 TSO model (which are similar).

Finally, to ensure soundness, we add a designated flag $overflow$ to track when the buffer size grows beyond our predetermined bound $k$. Overall, with a bound of $k = 1$, from the 8 predicates used for sequential consistency, we generate 15 predicates for PSO:

\begin{align*}
(Msg = 0), (Ack = 0), (lSSSt = 0), (lAck = 0), (lMsg = 0), (lRSt = 0), (lRCnt = lSCnt), \\
(lRCnt + 1 = lSCnt), (Msg_{\cdot cnt} \cdot 0 = 0), (Msg_{\cdot cnt} \cdot 0 = 1), (Ack_{\cdot cnt} \cdot 1 = 0), \\
(Ack_{\cdot cnt} \cdot 1 = 1), (Msg_{\cdot 1} \cdot 0 = 0), (Ack_{\cdot 1} \cdot 1 = 0), (overflow = 0).
\end{align*}

We refer the reader to [DMVY13] for details of the Cube Extrapolation part, which was a crucial part of the work and was enabling: made it possible to verify larger algorithms. The extrapolated predicates (and cubes) are then used to build the Boolean Program and makes it possible to proceed to the next step.

**Step 4: Model checking** Once the boolean program is built (either via Predicate extrapolation or Cube Extrapolation), the final step is to model check the program. This step is exactly the same as in the standard case of traditional predicate abstraction, shown in Fig. 3.3(a).

For details and results of this approach, we refer the reader to [DMVY13].
Chapter 4

Numerical Abstraction for Relaxed Memory Models

As a natural extension of Chapter 3, we attempted to verify programs running under RMMs using numerical abstractions [CH78]. We explored it as an additional option for applying extrapolation. We used program translation similar to the one from Chapter 3 and applied the ConcurInterproc[Jea09] tool on the original program under SC and on the result of our translation. We were able to find program-specification pairs for which a program could be verified under SC but not under RMM without synchronization (referred to in Chapter 7 as Assumption 1 and Assumption 2). But, before we could continue, our research took a different direction. We observed that a core reason why RMM verification failed was the nondeterminism added by our translation, which led us to the notion of verifying programs by instrumenting non-determinism. We thought that this notion could be applied to general verification, and we discovered an instance where indeed it could be (a version of the Bakery algorithm sketched in Fig. 1.1)

As with predicate abstraction for RMMs, our verification approach with numerical domains has two parts:

- **Preform a code-to-code analysis**, encoding RMM effects, to create a new program $P_{RMM}$ on which an existing Sequential Consistency verifier can run, and which exhibits (when run under SC) all the behaviors of $P$ (when run under RMM).

- **Run the verifier and assist it** in carrying out numerical abstraction analysis, using an observation that will be explained shortly.

Numerical abstraction analysis with Box, Octagon, or Polyhedra domains is an efficient verification technique but it has one drawback – it can not represent disjunctions. To explain this drawback we will use the simple fragment of code in Fig. 4.1. After execution of the code segment of Fig. 4.1, the value of $Y_1$ depends on whether the if then branch has been taken. A precise abstraction analysis will capture the value of $Y_1$, after the join at the endif, as $Y_1 == t_1 \lor Y_1 == -t_1$. Convex numerical domains
Y1 = -t1;
if nondet then
  Y1 = t1;
endif;

Figure 4.1: A non-determinism example

Thread 1:
1: flag0 = 1;
2: turn = 1;
3: f1 = flag1;
4: lt1 = turn;
5: if ((lt1 != 0) & (f1 != 0))
6: goto 3;
7: nop; // CS
8: flag0 = 0;
9: goto 1;

Thread 2:
1: flag1 = 1;
2: turn = 0;
3: f2 = flag0;
4: lt2 = turn;
5: if ((f2 != 0) & (lt2 = 0))
6: goto 3;
7: nop; // CS
8: flag1 = 0;
9: goto 1;

assert always ((pc1 ≠ line 7) ∨ (pc2 ≠ line 7))

Figure 4.2: Peterson’s mutual exclusion algorithm

such as Box, Octagon, and Polyhedra, cannot represent a disjunction because it is a
non-convex element. Thus, a join operation at the endif point may cause too much
information to be lost. We will receive an element $Y1 < - t1 \land Y1 >= - t1$ (assuming
t1 is positive). As t1 becomes larger, the value of Y1 becomes more imprecise. Our
code-to-code translation added constructs similar to the one in Fig. 4.1, with which
numerical analysis had difficulty coping.

We utilized the fact that ConcurInterproc\[Jea09\] tool stored separately a numerical
and a boolean part of the state. We added a boolean variable, indicating whether the
if branch had been taken. This proved to be an enabling approach. A caveat of this
approach was the abundance of locations where those boolean variables could be placed
and the verification, time and space, overhead it caused – every additional boolean
variable might double the number of states during verification. We took our verification
attempts to the next step and implemented a search for the minimal verifiable placement
of fences and boolean variables.

4.1 Overview

In this section, we provide an informal overview of our approach using Peterson’s mutual
exclusion algorithm (shown in Fig. 4.2). More elaborate examples are considered in
Section 4.4.

4.1.1 Motivating Example

In Fig. 4.2, each of the two threads attempts to reach its critical section (CS) in line 7.
To enter the critical section, a thread first checks whether the other thread intends to
enter the critical section (by checking the value of flag0 or flag1), as well as whether it is its turn (by checking the value of turn).

Our goal is to guarantee that both threads do not enter the critical section simultaneously. This property holds when the two threads run on a sequentially consistent machine, but no longer holds when they run on a Relaxed Memory Model such as PSO or TSO. Under Relaxed Memory Models, the writes to flag0, flag1, and turn performed by one thread may be buffered and not yet visible to the other thread when it reaches the condition in line 5. As a result, both processes may enter the critical section simultaneously. For example, if thread 1 enters the critical section, and its write to flag0=1 has not yet been flushed to main memory, thread 2 will pass its check in line 5 and also enter the critical section. To guarantee that mutual exclusion holds under Relaxed Memory Models, the programmer has to explicitly add memory fences to the program. However, because fences are expensive, the programmer faces the challenge of inserting the minimal set of fences required to ensure mutual exclusion.

4.1.2 Searching for Fence Assignment and Refinement Placement

Our goal is to synthesize a minimal fence assignment for a given program, specification, and memory model. Finding such a minimal fence assignment involves a search over the space of possible fences and automatically checking the correctness of each program in the space. To automatically verify a program, we employ abstraction refinement. In our setting, abstraction refinement is described as a set of program locations (discussed in detail later) which we refer to as a refinement placement. This leads to the following two-dimensional synthesis challenge:

Find a refinement placement and a minimal fence assignment which verify the program.

Naive approach A naive approach where we perform an exhaustive search of the fence/refinement space is almost always non-feasible. For example, even for Peterson’s algorithm, there are $2^6$ potential fence assignments and $2^{23}$ potential refinement placements (we explain these in Section 4.1.3), leading to a total number of $2^{29}$ points in the fence/refinement space!

Our approach: semantic program and proof propagation Our approach works by pruning large parts of the search space, based on the following two observations (here we use the notation $P(f, r)$ to mean program $P$ with fence assignment $f$ and refinement placement $r$):

- **Implied correctness**: if the program $P(f, r)$ is verified successfully, then it implies the correctness of any other point in space which uses a superset of the fences in $f$ or a superset of the refinement locations in $r$.

- **Implied incorrectness**: if the program $P(f, r)$ fails to verify, then it implies the
propagation of:
program correctness
+ abstraction refinements

means the program need not be explored further

means the program has been explored

means

is a successful abstraction refinement used to verify program

is a suggestion to prove program with a combined abstraction refinement

Legend:

Figure 4.3: Propagation of program correctness and abstraction refinements.

incorrectness of any other point in space which uses a subset of the fences in \( f \) or a subset of the refinement locations in \( r \).

Fig. 4.3 shows an example of one of our propagation techniques (discussed in Section 4.2.1) and is meant to give the reader some intuition about how it works. Here, successful verification of \( P(f_1, r_1) \) and \( P(f_2, r_2) \) implies the correctness of all programs in the search space “below” these two (all programs with a subset of the fences). Further exploration of the space can first attempt to verify the point \( P(f_3, r_3) \) which employs a smaller set of fences \( (f_3 = f_1 \cap f_2) \) yet uses a refinement placement which combines successful refinement placements from different programs (i.e. \( r_1 \) and \( r_2 \)). The intuition behind this combination is that slight relaxation of the program via fewer fences should only require slight adjustment of the abstraction refinement. Our experimental evaluation (Section 5.4) shows that propagation is effective for finding a minimal fence assignment for many of our benchmarks.

4.1.3 Refinement Placement - Reduction & Abstraction

We next describe several ingredients of our approach to verification of infinite-state programs running on Relaxed Memory Models.

As described in the motivating example, on a Relaxed Memory Model, writes to shared memory are not immediately visible to all processes: writes are first placed into a local buffer and then (non-deterministically), a flush instruction pops values from that buffer and writes them to shared memory. In our setting, this mechanism is encoded in source code via a translation phase.

Non-determinism due to flushes Fig. 4.4 shows the translation of the assignment statement \( \text{flag}0=1 \) for the PSO memory model. (In this model, each thread maintains a FIFO buffer for each shared variable.) We can see that, \( \text{flag}0=1 \) is translated into two parts: i) the write to the FIFO buffer and a non-deterministic \( \text{flush} \). Details of
/* begin store */
if flag0_cnt_0 > 0 {
    overflow = true;
    halt;
}
flag0_cnt_0 = flag0_cnt_0 + 1;
if flag0_cnt_0 < 2
    flag0_1_0 = 1;
/* end store */
/* begin flush */
yield;
while * do {
    if flag0_cnt_0 > 0 {
        flag0 = flag0_1_0;
        flag0_cnt_0 = flag0_cnt_0 - 1;
        yield;
    }
}
/* end flush */
yield;

Figure 4.4: Translation of flag0=1 under PSO.

this translation are discussed in Section 4.3. Here, we only discuss the translation of
the flush. A flush from a store buffer works by writing back to shared memory an
arbitrary number of items from the buffer. This is captured by the while (*) loop
which has a non-deterministic termination condition (denoted by *).

The non-deterministic loop introduces a significant challenge when reasoning with
numerical domains (which capture state via relations between variables). The reason
is that two program states, appearing right after the while loop has completed, differ
significantly depending on whether the flush was performed. Both states can be
captured with disjunctions, but standard (convex) numerical domains often dramatically
lose precision exactly in such (disjunctive) cases.

Local abstraction refinement To address this loss of precision, we use an abstract
domain that combines numerical information with a finite boolean domain. By carefully
introducing boolean predicates, we can refine the abstraction (by splitting the numerical
state) in a local manner. While local refinement may restore sufficient precision for
successful verification, it unfortunately comes at an exponential cost. The addition
of new predicates can lead to an exponential blowup of the program analysis, as each
predicate may double the state space. Furthermore, such a refinement is not required
for all locations of a flush. For example, if we can prove that a flush is always reached
with an empty buffer, the flush will have no effect, and thus there is no need to refine
the abstraction at such locations (we elaborate on this point in Section 4.3.4).

This introduces the challenge of finding a suitable refinement placement (a subset of
the flush program locations) that is precise enough to enable verification yet is scalable
enough for the analysis to terminate in reasonable time.
4.2 Abstraction-Guided Fence Synthesis

In this section, we present a new synthesis algorithm which propagates both fence assignments and refinement placements. Our algorithm leverages implied correctness/incorrectness to reduce the search space. The algorithm treats the two dimensions of the problem as having the same importance, and looks for a minimal fence assignment and a minimal refinement. In addition, the algorithm strives to minimize the number of fences based on a new concept where an abstraction refinement is obtained by combining successful refinements across programs.

4.2.1 Abstraction-Guided Fence Synthesis

Algorithm 4.1 provides a declarative description of our approach. The algorithm takes as input a program $P$, a specification $S$, a memory model $M$ and an abstraction $\alpha$, and produces a (possibly modified) program $P'$ that satisfies the specification under $M$ with a minimal verifiable fence assignment. The algorithm leverages information from several points in the space in the verification effort of a given point.

**Fence assignment and refinement placement** A fence assignment $f$ for a program $P$ with program labels $\text{Lab}_p$ is simply a subset of program labels $f \subseteq \text{Lab}_p$. A refinement placement $r$ for a program $P$ is also a subset of program labels, but it is restricted only to program labels of flush operations. The details of the refinement are elaborated in Section 4.3 and are not important for understanding the central concept of the synthesis algorithm presented in this section. For a given fence assignment $f$ and a refinement placement $r$, $P\langle f, r \rangle$ denotes the program $P$ with fences placed according to $f$ and an abstraction refinement selected according to $r$.

**Searching for satisfying placements** The algorithm begins by initializing a worklist with (i) the program under a full fence assignment (line 3) together with (ii) a refinement placement of program locations that are reasonable (line 4) as determined by our Empty Buffer Analysis (EBA) (see Section 4.3.4).

For each element of the worklist, the algorithm tries to improve the fence assignment and refinement placement (Lines 9 and 10). The operation of these two functions is discussed later in this section. Our algorithm then invokes the underlying verifier to check whether $[P\langle f, r \rangle]^\alpha_M = [S]^\alpha_M$ (line 11).

**Optimized Semantic Search** Our algorithm maintains the two sets, verified and falsified, for storing points $(f, r)$ that have been verified or where verification failed, respectively. Initially, both of these sets are empty. In the case of successful verification, the algorithm adds $(f, r)$ to the set of verified points in space. However, the algorithm does more than that: it also adds to verified all points which consist of a superset of fences as well as a superset of refinements. Successful verification of $P\langle f, r \rangle$ means that the search can proceed to explore more relaxed versions of the program. The helper function $\text{relax}(f, r, K)$ is used to compute a set of $(f', r')$ pairs that admit more
Algorithm 4.1 Semantic search for finding minimal verifiable fence assignments.

**Input:** $P$ - program, $S$ - Spec, $M$ - memory model, $\alpha$ - abstraction, s.t. $[P]_{SC}^\alpha \models [S]_{M}^\alpha$

**Output:** $P'$ - program such that $[P']_{M}^\alpha \models [S]_{M}^\alpha$ with minimal a number of fences

1: verified = $\emptyset$
2: falsified = $\emptyset$
3: $f = \text{fullFenceAssignment}(P)$
4: worklist = $\{(f, \text{EBA}(P,f))\}$
5: while worklist $\neq \emptyset$ do
6: $(f,r) = \text{select some pair from worklist}$
7: known = verified $\cup$ falsified
8: $f = \text{improveF}(f,\text{known})$
9: $r = \text{improveR}(f,r,\text{known})$
10: if $[P(f,r)]_{M}^\alpha \models [S]_{M}^\alpha$ then
11: verified $\cup$ $\{(f,\tilde{f})| f \subseteq \hat{f}, \hat{r} \subseteq \tilde{r}\}$
12: alternatives = $\text{relax}(f,r,\text{known})$
13: else
14: falsified $\cup$ $\{(f',r')| f' \subseteq f, r' \subseteq r\}$
15: alternatives = $\text{restrict}(f,r,\text{known})$
16: end if
17: worklist = (worklist $\cup$ alternatives) $\setminus$ known
18: end while
19: $(f,r) = \text{min}(\text{verified})$
20: return $P(f,r)$
behaviors (via a subset of fences) as well as coarser abstractions:

\[
\text{relax}(f, r, K) = \{ \langle f', r' \rangle \mid f' \subseteq f \text{ and } r' \subseteq r \text{ and } \langle f', r' \rangle \notin K \}
\]

In the case of failed verification, the algorithms can add \( \langle f, r \rangle \) to the set of falsified points in space, but once again, it can do more than that. That is, the algorithm adds to the set falsified all points in the space which consist of a subset of fences and a subset of abstraction refinements. Failed verification \( \langle f, r \rangle \) means that the search should explore more restricted versions of the program. The helper function \( \text{restrict}(f, r, K) \) computes a set of \( \langle f', r' \rangle \) pairs that admit fewer behaviors (via a superset of fences) as well as more refined abstractions:

\[
\text{restrict}(f, r, K) = \{ \langle f', r' \rangle \mid f \subseteq f' \text{ and } r \subset r' \text{ and } \langle f', r' \rangle \notin K \}
\]

The algorithm terminates when there are no more alternatives to explore, and returns a program with a minimal fence assignment. (In our implementation, we return all non-comparable minimal fence assignments.)

**Parametric choices** Our algorithm is parameterized on three dimensions:

- The choice for the next pair \( \langle f, r \rangle \) to select at line 7. The method of choosing the next element determines whether our search will be similar to a depth-first search, to a breadth-first search or to a search that explores random elements of the space.

- The function \( \text{ImproveF} \). This function leverages the knowledge of previous verification attempts (i.e., from the set known). For example, if \( f \) already verified for a different refinement \( r' \) and \( f' \) is a configuration in known which verified, then we can inspect \( f \cap f' \) instead of \( f \).

- The function \( \text{ImproveR} \). With this function we improve the refinement \( r \) based on available knowledge (i.e., from the set known). For a fence assignment \( f \) and a refinement \( r \), if \( \langle f', r \rangle \) and \( \langle f'', r' \rangle \) both previously successfully verified and if \( f' \) and \( f'' \) are stronger than \( f \) (that is, a superset of fences), then \( \text{ImproveR} \) will return \( r \cup r' \). This makes the abstraction refinement more precise, increasing the chances of success.

### 4.3 Automatic Verification

In this section we discuss the three steps of our automatic verification procedure: the reduction procedure, the underlying program analysis and the mechanism of abstraction.
refinement. We also discuss a static empty buffer analysis that is used by our algorithm to compute a set of possible abstraction placements to chose from (as discussed earlier).

### 4.3.1 Reduction

Similarly to [AKNT13] and Chapter 7, we reduce a program $P$ running on a relaxed model $M$ to a program $P_M$ running on sequential consistency. This enables us to directly leverage advances in program analysis for sequential consistency. We adopt a translation procedure similar to the one in Chapter 3, where the key idea in constructing $P_M$ is to represent the abstraction of the store buffers of $M$ as variables in $P_M$. We illustrate the process for when $M$ is the PSO memory model. The process for x86-TSO is similar.

For PSO, it is sufficient to consider a program $P_{PSO}$ where every shared variable $X$ in the program $P$ is also associated with: (i) additional $k$ local variables for each thread $t$: $x_{1,t}, \ldots, x_{k,t}$, representing the content of a local store buffer for this variable in each thread $t$, (ii) a buffer counter variable $x_{cnt,t}$ that records the current position in the store buffer of $X$ in thread $t$.

The translation uses the function $[\_\_\_]$ which takes as input a statement $S$, a thread $t$, and a bound $k$ on the maximum buffer size and produces a new statement as output $[S]_k^t$ (Fig. 4.5). The translation procedure is described in detail in Chapter 7. Let us take a closer look at the most challenging method, the $\text{flush}$.

A $\text{flush}$ is translated into a non-deterministic loop. If the buffer counter for the variable is positive, then it non-deterministically decides whether to update the shared variable $X$. If it has decided to update $X$, the earliest write (i.e. $x_{1,t}$) is stored in $X$. The contents of the local variables are then updated by shifting: the content of each $x_{i,t}$
is taken from the content of the successor $x_{(i+1),r}$ where $1 \leq i < k$. Finally, the buffer count is decremented.

In our encoding of concurrent programs, context switches between threads are explicitly specified with yield statements. (We place yield statements after every instruction.) Since under the Relaxed Memory Model a flush can be executed nondeterministically by the memory subsystem at any moment during program execution, our reduction places a (translated) flush after every yield.

4.3.2 Analysis with Numerical Abstract Domains

Once we have obtained the reduced program $P_M$, we use abstract interpretation with advanced numerical domains to verify its properties under sequential consistency. In particular, if the property we are interested in verifying relates only to shared numerical variables $G$ appearing in program $P$ (for example, the property of no array access out of bounds), then when translating accesses to variables of $G$ by $P$, the reduction to program $P_M$ will only introduce additional numerical variables over the variables in $G$: these are the local variables and counters. This enables us to directly use powerful numerical domains such as the Polyhedra abstract domain over the resulting program $P_M$. There are three possible outcomes of the automatic verification step:

- The program $P_M$ verifies, in which case the verification is successful.
- The program $P_M$ does not verify because an overflow occurred during the analysis. There could be two reasons why overflow occurs:
  1. There exists a concrete execution in the program which indeed leads to an overflow (e.g. multiple stores to a shared variable without a fence in between).
  2. The abstraction is imprecise enough to establish that there is no overflow.

We cannot distinguish between these two cases and hence, when overflow occurs, we increase $k$ to a small bound (at most the number of removed fences) or refine the abstraction (detailed below). Our experience is that small values of $k$ combined with an abstraction refinement of the numerical analysis work well in practice.

- The program $P_M$ does not verify because the property being checked fails to verify under the current abstraction. In this case, we typically apply abstraction refinement to the numerical analysis.

4.3.3 Abstraction Refinement of Numerical Analysis

As discussed above, abstraction refinement is often a vital step to enable successful verification of the program $P_M$. A key question then is which parts of the program $P_M$ require a more refined treatment in the abstract? To find these statements in $P_M$, we employ a two-step approach, where we always first verify the program $P$ under
sequential consistency, before trying to verify the translated program $P_M$. This allows us to focus the search for abstraction refinement on the statements in $P_M$ that are the root cause for the new behaviors. In our setting, these are the flush instructions appearing in $P_M$, as it is via these statements that relaxed memory effects eventually become visible.

Abstraction refinement of the numerical analysis is accomplished in our system by directly encoding the suggested refinement into the program $P_M$ by automatically introducing boolean auxiliary variables at places where the memory model relaxation takes effect. In particular, the number of boolean variables is proportional to $k$ and these boolean variables are initialized appropriately inside the branches of the translated flush statement (to true or false respectively, depending at which branch the boolean variable is assigned). For example for Peterson’s algorithm (Fig. 4.2), under minimal verifiable fence placement for TSO, our analysis found that a boolean variable was needed at the flush after Thread 2 assignment to turn but not after that assignment for Thread 1.

This is yet another advantage of the reduction approach: it enables us to quickly experiment with and provide the abstraction refinements over the base numerical domains by modifying the program $P_M$ instead of trying to somehow change the internals of an existing program analyzer (or build a new analysis). In particular, our system integrates with ConcurInterproc [Jea] (which supports logico-numerical domains), enabling us to match the auxiliary boolean variables with the logical part of the combined domain.

Overall, we believe that we have found a good match between the particular type of abstraction refinement required in our context, the fact that this refinement can be encoded in the program, and the ability of an existing analyzer to incorporate this encoding directly into its abstract domain.

4.3.4 Empty-Buffer Analysis

The additional predicates from refinement placement track the non-determinism due to flushes. However, such non-determinism is only relevant when the store buffers are not empty. When the store buffers are empty, a flush operation has no effect, and thus there is no need for a refinement at that program point. The challenge of course is to statically identify program locations in which the store buffers are guaranteed to be empty in any possible execution of the program. Towards that end, we use a simple static analysis that identifies program points where buffers are guaranteed to be empty. The analysis is sound, when it reports that a store buffer is empty, it guarantees that it will be empty in any possible execution. In Section 5.4, we show that the empty buffer analysis is effective and produces an upper bound on refinements that is significantly lower than the total number of possible locations.
4.4 Evaluation

We implemented our approach as described in previous sections and evaluated it on a range of challenging concurrent algorithms. To the best of our knowledge, this is the first extensive analysis study in the context of Relaxed Memory Models involving infinite-state reasoning and abstract interpretation. All of our experiments were performed on an Intel(R) Xeon(R) 2.13GHz machine with 250 GB RAM.

For the automatic verification step, we used ConcurInterproc [Jea13], which uses the APRON numerical abstract domain library [JM09]. To check that the inferred invariants imply the specification, we used the Z3 SMT solver.

4.4.1 Concurrent Algorithms

In our experiments we used 15 concurrent algorithms (7 finite-state and 8 infinite-state). Among these, there are 3 (infinite-state) array-based work-stealing queues and 7 mutual exclusion algorithms. We are not aware of any previous attempts to automatically verify properties of concurrent data structures such as the work stealing queues (WSQs) under relaxed models. For all of the algorithms we verified safety properties (e.g. a pair of labels is unreachable). For the WSQs, we verified consistency properties such as: the head index of the queue is always less than the tail index.

While our technique never reports incorrect fence assignments, the final result might lose the minimality guarantee due to non-monotonic analysis. We note that in our benchmarks, this situation was never encountered. To cope with non-monotonicity, the tool has to spend more time searching when intermediate points fail to verify. In specific situations (certain outputs from ConcurInterproc), the search continues or even reattempts to verify a program when ConcurInterproc returns “unknown”.

4.4.2 Results

Our experimental results for both PSO and TSO memory models are summarized in Tables 4.1, 4.2, 4.3, and 4.4. The first column of each table contains a tuple, under each benchmark name – the first element is the maximal number of fences for the algorithm, and the second element is the total number of locations for abstraction placements. For each benchmark, we bounded the search time to an hour, two hours and four hours. Each time bound result has two parts: the minimal number of fences achieved (columns labeled $f$) and the minimal relaxation under that fence assignment that the algorithm was able to find (columns labeled $r$). We compared three versions of our search:(i) breadth-first search (lines labeled BFS), (ii) depth-first search (lines labeled DFS) both without propagation and (iii) search with propagation (prop). At each point, the algorithm explores the next element from the worklist which is highest in the lattice for BFS, or lowest for DFS. After successful or failed verification and updating the set known, we update the worklist with the immediate successors of the
Table 4.1: PSO results. The graphs show discovered fence assignments over time.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>max(f, r)</th>
<th>1h</th>
<th>2h</th>
<th>4h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>r</td>
<td>f</td>
</tr>
<tr>
<td>abp</td>
<td>(2,17)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>concloop</td>
<td>(4,14)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>dekker</td>
<td>(10,34)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>10</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>10</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>9</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>kessel</td>
<td>(6,12)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>loop2-TLM</td>
<td>(6,21)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>pc1</td>
<td>(9,27)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>9</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>psql</td>
<td>(8,23)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>8</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>sober</td>
<td>(11,23)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>11</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: PSO results. The graphs show discovered refinement placements over time.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>max (f, r)</th>
<th>1h</th>
<th>2h</th>
<th>4h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>r</td>
<td>f</td>
</tr>
<tr>
<td>wsq-chase</td>
<td>(4,19)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>wsq-fifo</td>
<td>(2,13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>wsq-the</td>
<td>(7,33)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>7</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>7</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Algorithm</td>
<td>1h</td>
<td>2h</td>
<td>4h</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>abp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2,17)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>conloop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,14)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>kessel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,12)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>loop2-TLM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,21)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>pc1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(9,27)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>9</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>5</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>peterson</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,23)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>6</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>postgresl</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(8,23)</td>
<td>f</td>
<td>r</td>
<td>f</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.3: TSO results. The graphs show discovered fence assignments over time.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1h</th>
<th>2h</th>
<th>4h</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,13)</td>
<td>f</td>
<td>r</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>wsq-chase</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,19)</td>
<td>f</td>
<td>r</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>prop</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bfs</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>dfs</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: TSO results. The graphs show discovered refinement placements over time.
attempted configuration (above or below the explored element - depending on whether it was verified). The third search configuration (labeled \textit{prop}) is a BFS search \textit{with propagation}.

The graphs depict the time it took to discover the minimal fence assignment. The x-axis is the time in an "hour:minute:seconds" format and the y-axis is the number of fences discovered. For some cases, such as \texttt{pc1}, it can be seen that the initial behavior of the \textit{prop} approach is similar to that of DFS. This is due to a "streak" of successful verifications where a successful verification from a previous stage (say fence assignments "remove #9" and "remove #8" verified) affects the next element attempted by the \textit{prop} approach (for the example given "remove #8 and #9" will be attempted). This behavior is similar to DFS. For algorithms such as \textit{kessel} and \textit{pgsql}, it can be seen that the DFS approach finds a non-optimal fence assignment early in the search (the \textit{prop} approach finds a better assignment later), and no new points appear in the graph. In those cases the DFS approach proceeds to explore lower elements in the lattice and fails repeatedly.

For several algorithms only the full assignment of fences was verified. Those algorithms are described in Table 4.2. Here (unlike Table 4.1), the y-axis is the abstraction refinement placement that verified. Those graphs have more points and describe more clearly the difference between the three approaches (DFS, BFS and \textit{prop}). In many cases, BFS explores "too many" elements high in the lattice, while DFS converges fast to the lowest element it can verify but then it needs to backtrack. For \texttt{wsq-the}, DFS didn’t find a placement smaller than the full one. Given more time, perhaps it would "backtrack" and find a placement equivalent to the one the \textit{prop} approach found.

\textbf{Summary} It can be seen that the search with propagation (\textit{prop}) finds smaller fence assignments quicker than BFS and fewer or equal fence assignments than DFS.

\section{4.5 Related Work}

Next, we discuss some of the work most closely related to ours. This includes work on automatic verification techniques, dynamic analysis and bounded model checking approaches, search propagation in synthesis, as well as robustness. Generally, while some work has been done on bounded model checking of concurrent programs running on Relaxed Memory Models, almost none has been done on automatically verifying infinite-state concurrent programs running on these models.

\textbf{Program transformation} One general direction for handling Relaxed Memory Model programs is to encode their effects into a program and then analyze the resulting program using standard tools geared towards sequential consistency. Towards that end, the works of Algave and Atig [AKNT13, ABP11] suggest source-to-source transformations which encode the relaxed memory semantics into the target program. We also believe that this is a viable path and in our work, we also use a similar encoding approach. However,
as we have seen, direct encoding of the semantics is typically not sufficient when dealing with infinite-state programs where it is critical that the abstraction be precise.

**Handling infinite-state programs** Kuperstein et al. [KVY11] (which was also described in Chapter 3) handle some forms of infinite-ness (such as that coming from the buffers), but do not handle general infinite-state programs under sequential consistency. Other works in this direction are those of Linden et al. [LW10, LW13] which show how to use automata as symbolic representations of store buffers. Their work focused on programs that are finite-state under sequential consistency. Vafeiadis et al. [VN11] present an approach for eliminating fences under x86-TSO. Their approach is based on compiler transformations and assumes that the input program is correct. The work of Abdulla et al. [AAC+12] (which was also described in Chapter 3) builds on [KVY11] and is able to handle infinite-state programs under x86-TSO. That work combines predicate abstraction with the store buffers abstraction from [KVY11]. The approach uses traditional abstraction refinement in order to discover the necessary predicates. [DMVY13] (which was also described in Chapter 3) handles both x86-TSO and PSO memory models and also uses predicate abstraction. However, the procedure for inferring the predicates necessary to verify the program under Relaxed Memory Models differs from standard abstract refinement. Instead, the paper proposes a form of proof extrapolation: it first assumes that the program is verified under sequential consistency and then shows how to adapt these predicates (in a memory-model specific way) into new predicates which are then used as candidates for the verification under the particular Relaxed Memory Model. Both [DMVY13] and [AAC+12] approaches are based on predicate abstraction and require the predicates to be inferred via refinement or adaptation. In contrast, the techniques presented in this work are based on iterative numerical abstract interpretation, which promises to scale better (but is focused on numerical domains). In addition, our search algorithm combines propagation of abstraction refinements across programs with program restriction via fence inference. Our work also has relevance to the well known technique of lazy abstraction [HJMS02], which introduces the concept of adjusting the level of abstraction for different sections of the verified program’s state space. In our approach, the search can be seen as selectively introducing refinements that guide the analyzer. However, unlike previous work, we learn new refinements by combining existing successful refinements from several programs.

**Explicit model checking for Relaxed Memory Models** Several works (e.g., [KVY11, Jon09, KVY10, HR07]) have focused on explicit-state model checking under Relaxed Memory Models. Among those, [KVY10] focuses on fence inference and [HR07] also describes an explicit-state model checking and inference technique for the .NET memory model. These approaches are sound only for finite-state programs, and cannot handle infinite-state programs. CheckFence [BAM07] takes a different approach: instead of working with operational memory models and explicit model-checking, the technique converts a program into a form that can be checked against an axiomatic model spec-
ification. This technique unrolls loops at a preprocessing stage and cannot handle infinite-state programs.

In addition, there has recently been interest in exploring dynamic techniques for testing programs running on various Relaxed Memory Models. The work of Liu et al. [LNP+12] dynamically analyzes (via a demonic scheduler) concurrent algorithms under the TSO and PSO memory models and, whenever it finds a violating trace proposes a repair that inserts memory fences into the program. Recent work by Norris and Demsky[ND13], leverages various partial order reduction techniques for bounded model checking of concurrent C++ programs. Both of these work and that of Liu et al. attempt to handle larger programs by sacrificing soundness.
Chapter 5

Effective Abstractions for Verification under Relaxed Memory Models

During the work presented in Chapter 4 we observed that the abstraction and the translation phases of our process could benefit from each other. We moved the abstraction from being strictly numerical to being boolean-numerical, thus allowing finer control of joins of states and greater precision during the abstraction.

Utilizing the boolean abstraction allowed us coarser translation. More specifically, the change in one challenging translation structure – the flash operation – eliminated much of the precision loss.

Our Approach We present a new analysis system for verifying concurrent programs running on Relaxed Memory Models such as Intel’s x86 TSO and PSO buffered memory models. Our approach relies, first, on a new abstraction of the memory model that eliminates some of the expensive work in managing the store buffers, thus significantly reducing the analysis effort and improving its precision. This abstraction is also directly applicable and useful for other verification frameworks, both finite and infinite-state (e.g., bounded model checking, abstract interpretation and predicate abstraction). Our approach also leverages knowledge of the particular program analysis used in this work (abstract interpretation with numerical domains) to encode the size of the store buffers in a way that reduces the loss of precision under that abstract domain.

We incorporate the above two ideas into a robust analyzer. The analyzer uses a source-to-source transformation that enables direct use of existing program analyzers under sequential consistency for verifying concurrent programs running on Relaxed Memory Models. That is, given a program $P$, a specification $S$ and a memory model $M$, the transformation automatically constructs a new program $P_M$ such that if $P_M \models_{SC} S$ then $P \models_M S$. The program $P_M$ contains an abstraction of the relaxed behaviors induced by $M$, thereby ensuring the soundness of the approach.
initial values: X=0  Y=0

Thread 1:
1: X = 1
2: a = Y
3: X = a + 1
4: fence
5:

Thread 2:
1: Y = 1
2: b = X
3: Y = b + 1
4: fence
5:

Spec: ((pc1 = 5) ∧ (pc2 = 5)) ⇒ (X + Y ≥ 2)

Figure 5.1: Example program

While prior works [AKNT13], as well as Chapters 4 and 3, suggest source-to-source transformations, we show experimentally that the approach taken in this chapter is more precise and efficient: it enables verification of (infinite-state) concurrent algorithms that prior work cannot verify, and for programs where prior work succeeds, our approach is faster and requires less memory. This work also represents one of the few studies on using abstract interpretation for verifying properties of infinite-state concurrent programs running on Relaxed Memory Models. Moreover, our approach requires no user annotations.

Main contributions The main contributions of this chapter are:

- A new abstraction for the store buffers of the memory model that eliminates expensive shifting of buffer contents. This abstraction reduces the workload on subsequent program analyzers and improves their scalability and precision.

- A source-to-source transformation that realizes the new abstraction (and the memory model effects), producing a program that can be soundly analyzed with verifiers for sequential consistency. The translation also leverages knowledge of the underlying abstract domain in order to encode the size of the store buffers in a way which reduces the overall loss of analysis precision.

- A complete implementation of the approach integrated with ConcurInterproc [Jea13], a tool based on abstract interpretation [CC77] with numerical abstract domains that can analyze infinite-state concurrent programs under sequential consistency.

- A thorough empirical evaluation on a range of challenging concurrent algorithms. Experimental results indicate that our technique is superior in both precision and efficiency to prior work and enables verification, for the first time, of several concurrent algorithms running on Intel’s x86 TSO and PSO memory models.
5.1 Overview

In this section we illustrate our approach on a running example. The goal is to give some intuition about and informal understanding of the work. Full formal details are provided in later sections.

To understand our approach, consider the concurrent program shown in Fig. 5.1. It consists of two threads that share the integer variables $X$ and $Y$. (Variables $a$ and $b$ are local to each thread.) The figure also shows an assertion which holds once both threads have completed their execution, namely that $X + Y \geq 2$. Our objective is to verify that the program satisfies this assertion under Relaxed Memory Models such as Intel’s x86 TSO and PSO.

5.1.1 Relaxed Behaviors

In the example in Fig. 5.1, Thread 1 can execute the statements at labels 1 and 2 in the opposite order. Similarly, Thread 2 can execute the statements at labels 1 and 2 in the opposite order due to the nature of Relaxed Memory Models such as TSO. Relaxed models such as TSO allow program statements to be executed out of order, resulting in behaviors not possible under sequential consistency. Under TSO, a store and a load (by the same thread) accessing different memory locations are allowed to be reordered. Therefore, after both threads execute the statements at labels 1 and 2, one can end up in the state $X = Y = 0$. This state is impossible to obtain under sequential consistency (SC). Weaker models such as PSO allow not only the reordering of store and load instructions but even the reordering of two stores (if they access different memory locations). In general, such reorderings are possible because the processor maintains store buffers per thread, and delays expensive writes to shared memory. For instance, in Intel’s x86 TSO, every thread updates a FIFO store buffer where the thread enqueues its shared memory writes and the memory sub-system dequeues these buffered writes (in the order of least recent write first) non-deterministically and updates shared memory.

5.1.2 SC Equivalence vs. Flexible Safety Specifications

When considering the problem of verifying programs running on relaxed models, there are two general choices for how we select the safety property to be verified, each influencing the design of the analysis abstraction. One direction is to develop analyzers that try to prove and (if need be) enforce that the relaxed program produces results equivalent to the sequentially consistent program (and, if not equivalent, to insert fences that make it so). This line of work was pioneered by Shasha and Snir [SS88], with various works later improving on the precision of the analysis and fence inference [SFW+05, AKNP14].

Another direction, and the one pursued in this paper, is to develop analyzers which can enforce arbitrary safety properties, not only equivalence. This is advantageous for two reasons:
The relaxed program might produce behaviors which are valid yet do not exist under SC, and enforcing equivalence leads to generation of redundant fences. We will use the program in Fig. 5.1 to illustrate this point. As mentioned before, the state $X = Y = 1$ is reachable under TSO at the end of the program. This state is impossible to reach under SC. To achieve SC equivalence, additional fence statements should be inserted in the program to prevent re-orderings that lead to this state. If we focus on ensuring the safety specification, only the current fences at labels 4 in the two threads are sufficient for verification; and

(ii) even if equivalence is the right specification, it may be difficult to produce an analysis that can prove equivalence; if we write a more program-specific, flexible safety property (which enforces the same constraints), that property may be easier to verify. We illustrate this point in Section 5.5: we show that [AKNP14] produces redundant fences, which our analysis avoids.

5.1.3 Our Approach

We now discuss our technique, step by step, starting with our source-to-source transformation with the abstraction embedded in it.

Step 1: Buffer analysis

A preliminary step of our approach is a buffer-size analysis of the input program. (Recall that a buffer exists in each thread.) This analysis outputs an over-approximation of the size of the write buffer at each point in the program. For our running example, the analysis determines that at line 1 (of both threads), the maximum write buffer size is 1, at line 3 the maximum buffer size is 2, and at line 5, the maximum buffer size is 0 (due to fence).

Step 2: Abstraction and source-to-source transformation

Here, the write buffer of each thread is directly encoded into the source code of the target program. The transformation proceeds by processing the original program in a statement by statement manner. In Fig. 5.2, we show the result of applying our transformation for TSO on the statements of Thread 1. We next informally discuss this procedure.

To encode the store buffers used by the Relaxed Memory Model, we introduce two kinds of variables. An example of the first kind is $X_{t1}$, which captures the value of the first write to shared variable $X$ found in the buffer of thread $t_1$. An example of the second kind is the boolean variable $flagX_{t1}$, which denotes whether or not the first element of the write buffer of $t_1$ stores a write to shared variable $X$ (as in general the first write found in the buffer of thread $t_1$ could be to some other shared variable).

Returning to our example, since the buffer is initially empty, the statement $X = 1$ is translated to two updates. First, the new variable $X_{t1}$ is updated and set to 1, and second, the boolean variable $flagX_{t1}$ is set to true.

However, simply updating the two newly generated variables is not enough because under TSO (and PSO), the memory sub-system can trigger a non-deterministic flush of
Figure 5.2: The result of applying our transformation for TSO on Thread 1 from Fig. 5.1. The flush statements are not part of the program but need to be captured by the translation (and are inserted after every program statement).
a thread’s store buffer at any point. (The flush operation dequeues the least recent write in the buffer and updates shared memory with that write.) To capture this behavior, we add a special flush code fragment after every program statement. Therefore, in our example, a flush is added after the statements at labels 1, 2 and 3. The flush code fragments following the statements at labels 1 and 2 are identical. The loop captures the non-deterministic effects of the flush semantics: either the flush takes place and the write stored in \( X_{1t_1} \) is flushed to shared memory (and if so, the boolean variable \( \text{flag}X_{1t_1} \) is reset to \( \text{false} \)), or the program continues without changing the shared memory.

Statement \( a = Y \) is translated without change as the buffer size analysis determines that \( Y \) is never written to by Thread 1 and hence the value is always read from shared memory (as opposed to being read from the buffer).

Next, statement \( X = a + 1 \) is translated. The generated code fragment first tests whether \( \text{flag}X_{1t_1} \) is set to \( \text{true} \). This answers the question of whether the first position in the buffer is already taken. We need this test as it is statically unknown whether a non-deterministic flush has fired. Depending on the result of the test, we now know where to write the value \( a + 1 \). If the first position of the write buffer is occupied, \( a + 1 \) is stored to the second element of the write buffer and the appropriate flag is set (i.e., \( \text{flag}X_{2t_1} \) is set to \( \text{true} \)). Otherwise, we store the value \( a + 1 \) to the first position in the buffer and set the appropriate flag.

We next generate the flush code fragment after the statement at label 3. This flush code is slightly different than the previous two flush fragments because at this point in the translation, the buffer-size analysis indicates that the maximum possible buffer size is 2. Therefore, we need to dynamically check what the actual size of the buffer is and flush the appropriate entry. This can either be the variable \( X_{1t_1} \) or the variable \( X_{2t_1} \). Naturally, once the write to shared memory is completed, we set the corresponding auxiliary boolean variable accordingly: \( \text{flag}X_{1t_1} \) or \( \text{flag}X_{2t_1} \).

The flush operation removes elements from the buffer in FIFO order. The buffer content is represented as local variables \( (X_{1t_1}, X_{2t_1}, \ldots) \). Removing the first element \( (X_{1t_1}) \) would normally (see [DMVY13], Chapter 4) be followed by a shift of the remaining elements from the buffer \( (X_{1t_1} = X_{2t_1}, \ldots) \). A key point is that we do not shift the store buffer contents on flush (buffer shifting details in Section 5.2.3) as a direct encoding of the memory model would do (and as previous approaches do; see [DMVY13], Chapter 4). Doing less work on a flush leads to more precise analysis and greater efficiency than prior work.

Finally, the fence statement at label 4 ensures that all writes before the fence are flushed to shared memory. An assume statement on both boolean variables captures this requirement.

**An example trace** In Fig. 5.3 we illustrate how a particular program trace updates the shared memory and the newly generated variables. The first line of that figure shows the sequence of statements in the trace. The second line shows the shared memory
Figure 5.3: The effect of a program trace on shared state and the state used by the two translations. The figure shows only statements of Thread 1 as well as flushes affecting Thread 1’s write buffer.

state (before and after each statement is executed). The third line (titled “robust buffer abstraction”) shows the values of the newly generated variables. Here, the first square box corresponds to \(X_{1,t_1}\) and the second square box corresponds to \(X_{2,t_1}\). Similarly, the first flag corresponds to \(flagX_{1,t_1}\) and the second flag to \(flagX_{2,t_1}\). If a flag is raised, it means the variable is set to true; otherwise it is set to false. For now we can ignore the fourth line (this is a previous transformation used by [DMVY13] and Chapter 4 and is discussed in Section 5.3). We show and discuss the following trace:

(i) initially, \(flagX_{1,t_1}\) and \(flagX_{2,t_1}\) are set to false and shared variables \(X\) and \(Y\) contain 0;

(ii) thread 2 executes \(Y = 1\) and a flush updates \(Y\) in shared memory (the trace in Fig. 5.3 starts after this step);

(iii) thread 1 executes \(X = 1\), resulting in \(flagX_{1,t_1}\) being set to true and \(X_{1,t_1}\) containing the value 1;

(iv) thread 1 reads \(a = Y\), obtaining the value 1 (Fig. 5.3 omits local variable \(a\), so no changes are shown);

(v) thread 1 executes \(X = a + 1\) resulting in \(flagX_{2,t_1}\) being set to true and \(X_{2,t_1}\) containing the value 2, at which point we have two writes in the store buffer of Thread 1;

(vi) a flush of Thread 1’s buffer results in \(X_{1,t_1}\)’s value being written to shared memory, setting \(X\) to 1, and \(flagX_{1,t_1}\) is set to false to mark that the flush completed;

(vii) a flush of Thread 1’s buffer results in \(X\) being set to 2 in shared memory and in setting \(flagX_{2,t_1}\) to false.

Step 3: Program analysis Once the translated (and potentially infinite-state) concurrent program is obtained, the final step is to analyze it and attempt to prove the property of interest. Any analysis can be used; in this work we chose logico-numerical
abstract domains for the following reasons: (i) there are readily available tools that implement these domains (e.g., we use ConcurInterproc, which implements convex numerical domains combined with boolean values), allowing us to focus on the novel parts of the work, and (ii) our benchmarks manipulate numerical variables and the properties we prove depend only on such numerical manipulations. We do note, however, that our abstraction can be useful in any setting, not just that of abstract interpretation.

The resulting analysis outputs invariants for each pair of thread locations. For instance, at labels 5, when both threads have completed, a fragment of the resulting invariant produced by the analysis is:

$$\neg \text{flag}X_{1.t_1} \land \neg \text{flag}X_{2.t_1} \land X \geq X_{1.t_1} \land X_{1.t_1} \geq 1 \land ...$$

This invariant contains both a boolean part, consisting of concrete values for the auxiliary variables \( \text{flag}X_{1.t_1} \) and \( \text{flag}X_{2.t_1} \), and a numerical part in the polyhedra numerical domain: \( x \geq x_{1.t_1} \) and \( x_{1.t_1} \geq 1 \).

Both auxiliary boolean variables are \texttt{false}, which corresponds to an empty write buffer for Thread 1. From the numerical inequalities, we conclude that \( x \geq 1 \). Similar constraints are obtained for the variables in Thread 2, allowing us to conclude that \( y \geq 1 \). Thus, we can conclude that the specification \( x + y \geq 2 \) holds when both threads terminate.

We note that for our running example, direct handling of write buffer contents as used in Chapter 4 fails to verify the specification, even though the program satisfies it. This is because a direct, shift-based handling causes precision loss during the abstract analysis using numerical domains (as explained in Section 5.2.5). In the next section, we formally present our abstraction and transformation, discuss how it compares to prior work, and show why it leads to more scalable and precise analysis.

### 5.2 Background

In this section we provide a brief review of previous direct encoding techniques as well as our preliminary buffer size bound estimate analysis and terms that will be useful for our new abstraction in Section 5.3.

#### 5.2.1 Buffer Analysis

The first step in our verification process is a preliminary buffer analysis. Its goal is to establish sound approximations of the buffer sizes at each program point. We perform the analysis for each thread independently. We use the same analysis for both PSO and x86 TSO, and we interpret the results of the buffer analysis accordingly for each model.

Given thread \( t \), all the shared variables have an initial buffer size 0. For each write to a global variable \( X \), we increment the value of the buffer size for \( X \) by one. The read statements do not modify the buffer sizes. Fence statements reset all the buffer sizes to
0. In the case of if conditions, we propagate the buffer sizes in each branch and join
the buffer sizes at the end of the branches by selecting the higher value for each shared
variable.

We handle the loops by using a constant as the maximum value for buffer sizes, and
if this constant value is surpassed by the size of a buffer, then that buffer size becomes \( \infty \). Given loops, the buffer analysis requires several iterations, and we stop when we
reach a fix-point. When the analysis is complete, the result is a function which, given a
shared variable \( X \) and a label \( l \), returns the upper bound of the buffer size of \( X \) at \( l \).

The results of the buffer analysis are used directly for the PSO model. In the case
of x86 TSO, we are interested in the sum of all buffer sizes at a given label, because x86
TSO has a single buffer per thread for all shared variables.

5.2.2 Direct Source-to-Source Encoding

Let \( \text{Prog} \) be the set of all programs, \( \text{RMM} \) be the set of Relaxed Memory Models (in
this paper \( \text{RMM} = \{ \text{x86 TSO, PSO} \} \)), and \( \mathbb{N} \) the natural numbers. The translation
mechanism can be seen as a function with the signature: \( T : (\text{Prog} \times \text{RMM} \times \mathbb{N}) \rightarrow \text{Prog} \)
where \( P \in \text{Prog} \) is an input program, \( M \in \text{RMM} \) is a Relaxed Memory Model, and
\( b \in \mathbb{N} \) is a bound on the buffer size.

*The meaning of buffer size bound* \( b \)  Key elements of the x86 TSO memory model
(and the PSO memory model) are the store buffers found between each thread and
shared memory. Given buffer size bound \( b \), the output of the translation is a new
program \( P_M \in \text{Prog} \) where \( P_M = T(P, M, b) \).

By construction, the behavior of \( P_M \) under sequential consistency semantics captures
the behavior of \( P \) under the relaxed model \( M \), with the exception of potentially
overflowing the store buffer. That is, if during the execution of \( P_M \) an attempt is made
to store more than \( b \) elements to the buffer, then the program \( P_M \) aborts.

If we manage to verify that \( P_M \) satisfies the specification (without aborting), we can
guarantee that \( P \) satisfies the specification under the memory model \( M \). If the program
\( P_M \) aborts, we may have to refine our model and retry verification with a larger buffer
size.

It is generally impossible to statically determine the maximal store buffer size
reachable during a program execution. However, in practice, static analysis can over-
approximate the maximal possible store buffer size. We distinguish two cases:(i) the
over-approximated value is finite. In this case, the buffer size over-approximation is
useful in optimizing the transformation procedure, and (ii) the over-approximated value
is unbounded. In this case, the transformation has a fixed buffer bound defined by the
user.
**Direct Translation**

We first discuss the intuitive, direct translation function which encodes the relaxed memory semantics into the program source code. This direct translation is used by prior works focusing on infinite-state verification ([DMVY13] and Chapter 4). We denote this translation by:

\[ T_D : (\text{Prog} \times \text{Rmm} \times \mathbb{N}) \rightarrow \text{Prog}. \]

In the following, we use \textit{Local} to denote the set of local variables (per thread) and \textit{Shared} the set of global shared variables. Expressions, both numerical and boolean, can refer only to local variables. Statements can read and write global variables. We use \textit{Stmt} to denote all statements.

The translation encodes relaxed memory store buffers using temporary variables. For each statement of \( P \in \text{Prog} \) we generate a code segment that captures the relaxed behavior of that statement. We define a transformation function at statement level:

\[ [] \in \text{Stmt} \times \text{Thread} \times \mathbb{N} \rightarrow \text{Stmt}. \]

**5.2.3 Direct TSO Translation**

The direct translation introduces new variables for capturing the effect of storing values into store-buffers instead of directly into main memory. For TSO (the translation for PSO is introduced in Section 5.2.4), the buffer is modeled with the following local variables:

- variable identifiers: \( \text{lhs}_1 t, \text{lhs}_2 t, \ldots, \text{lhs}_b t \), where \( b \) is the maximum size of the buffer. The identifier of a global variable is an integer – it stores an index of the shared variable to be written to shared memory.
- buffer content values: \( \text{rhs}_1 t, \text{rhs}_2 t, \ldots, \text{rhs}_b t \) – each stores the actual value to be written to shared memory.
- buffer counter: \( \text{cnt}_t \) takes values in the range \([0, b]\) – it stores the size of the buffer during execution.

Fig. 5.4 presents the rules of the direct translation. In the translation of each statement, the generated sequence of statements is atomic. An exception to that rule is the flush, in which only the inside of the generated loop is atomic and context switches are allowed between the loop iterations.

**Write to a global variable** \( [X = r]_b^t \): the store to a global variable \( X \) first checks whether it can exceed the buffer bound \( b \), and if so, the program aborts. Otherwise, the counter is incremented. The rest of the logic checks the value of the counter and updates the corresponding local variables. The global variable \( X \) is not updated and only local variables are modified.
if (cnt_t=b)
    abort("overflow")
    cnt_t = cnt_t +1
else if (cnt_t=1)
    r = rhs1t
else
    r = X

if (cnt_t=b)
    lhs1t = X
    rhs1t = r
else
    r = X

while random do
    assume (cnt_t = 0)
    □ ∀ X ∈ Shared : 
        if (cnt_t>0)
            (lhs1t=X)
            r = rhs1t
        □ end 
        if (cnt_t>1)
            lhs1t = lhs2t
            rhs1t = rhs2t
            ... 
            if (cnt_t=b)
                lhsb-1t = lhsb1t
                rhsb-1t = rhsh2t
                cnt_t = cnt_t-1 
                yield

Figure 5.4: Direct TSO Translation Rules of TD

Read from a global variable \([r = X]_b^t\): the load from a global variable \(X\) checks the current depth of the buffer and then loads from the corresponding local variable. When the buffer is empty (i.e., \(cnt_t = 0\)), or the variable has no occurrences in the buffer, the load is performed directly from shared memory.

Fence statement \([\text{fence}]_b^t\): the fence waits for the buffer to be empty before executing.

Flush procedure \([\text{flush}]_b^t\): the flush procedure is translated into a non-deterministic loop (we use random). If the buffer counter is positive and the entry at position 1 in the buffer (\(lhs1t\)) refers to \(X\), then the write value at position 1 (i.e., \(rhs1t\)) is stored in \(X\). The contents of the local variables are then updated by shifting: the content of each \(rhs_{j+1}t\) is moved to its predecessor \(rhs_jt\) where \(1 \leq j < b\). Finally, the buffer count is decremented.

To encode non-deterministic flushes of the memory sub-system, a flush procedure is added by the translation function to the output program. The role of the flush procedure is to soundly encode the possible non-deterministic flushes of the store buffer, triggered by the memory subsystem. Naively, a faithful translation of the flush action requires placing the flush code after each statement of the program that accesses shared memory. However, this can be optimized using a simple preliminary static analysis that finds cases where the store buffer is guaranteed to be empty (and thus no flush is needed), or guaranteed to be bounded by a fixed size (and thus the flush code can be simplified).

Trace Example Returning to Fig. 5.3 of Section 5.1, the last row of the figure (titled “Direct translation”) illustrates how a given trace is processed using the direct translation. The key here is the processing of the first flush statement, where the contents of the store buffer are explicitly shifted. As we will see next, such explicit shifting is in fact completely avoided by our new abstraction and subsequent translation.
5.2.4 Direct PSO Translation

In contrast with the TSO model we introduced in Section 5.2.3, the buffer for PSO is modeled separately for each shared variable (next we use \( x \) to stand for an arbitrary shared variable):

- buffer content values: \( x_1^t, x_2^t, \ldots, x_b^t \) — each stores the actual value to be written to shared memory.
- buffer counter: \( x_{\text{cnt}}^t \) takes values in the range \([0, b]\) — it stores the size of the buffer during execution.

<table>
<thead>
<tr>
<th>([X = r]^t_b)</th>
<th>([r = X]^t_b)</th>
<th>([\text{flush}]^t_b)</th>
<th>([\text{fence}]^t_b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ((x_{\text{cnt}}^t = b)) abort(&quot;overflow&quot;); (r = x^t)</td>
<td>if ((x_{\text{cnt}}^t = 0)) while (random) (\triangleright \forall X \in \text{Shared}: \text{assume})</td>
<td>if ((x_{\text{cnt}}^t &gt; 0)) (\triangleright \forall X \in \text{Shared}: \text{assume})</td>
<td>if ((x_{\text{cnt}}^t = b)) (\triangleright \text{end})</td>
</tr>
<tr>
<td>(x_{\text{cnt}}^t = x_{\text{cnt}}^t + 1)</td>
<td>(x_{\text{cnt}}^t = x_{\text{cnt}}^t)</td>
<td>(r = x_1^t)</td>
<td>if ((x_{\text{cnt}}^t &gt; 1)) (\triangleright \text{end})</td>
</tr>
<tr>
<td>if ((x_{\text{cnt}}^t = 1)) (r = x_0^t)</td>
<td>if (random) (X = x_1^t)</td>
<td>(x_{\text{cnt}}^t = 0)</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\triangleright \text{end})</td>
<td></td>
</tr>
<tr>
<td>if ((x_{\text{cnt}}^t = b)) (x_0^t = r)</td>
<td>if ((x_{\text{cnt}}^t = b)) (x_{(b-1)}^t = x_0^t)</td>
<td>(x_{\text{cnt}}^t = 1;)</td>
<td></td>
</tr>
<tr>
<td>(\triangleright \text{end})</td>
<td>(\triangleright \text{end})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5: Direct PSO Translation Rules of \(T_D\)

Fig. 5.5 presents the rules of the direct translation for PSO. Similar to the TSO case, in the translation of each statement, the generated sequence of statements is atomic, except the flush, in which only the inside of the generated loop is atomic.

**Write to a global variable** \([X = r]^t_b\): the store to a global variable \(X\) first checks whether the buffer of variable \(X\) can exceed the bound \(b\), and if so, the program aborts. Otherwise, the counter is incremented. The rest of the logic checks the value of the counter and updates the corresponding local variables. The global variable \(X\) is not updated and only local variables are modified.

**Read from a global variable** \([r = X]^t_b\): the load from a global variable \(X\) checks the current depth of the buffer of variable \(X\) and then loads from the corresponding local variable. When the buffer is empty (i.e., \(x_{\text{cnt}}^t = 0\)), the load is performed directly from shared memory.

**Fence statement** \([\text{fence}]^t_b\): the fence waits for all the buffers to be empty before executing.

**Flush procedure** \([\text{flush}]^t_b\): the flush procedure is translated into a non-deterministic loop (we use \(\text{random}\)). For each shared variable \(X\), if the buffer counter is positive for \(X\) (i.e., \(x_{\text{cnt}}^t > 0\)), then the write value at position 1 (i.e., \(x_1^t\)) is stored in \(X\). The
contents of the local variables are then updated by shifting: the content of each $X_j[t]$ is moved to its predecessor $X_{j-1}[t]$ where $1 \leq j < b$. Finally, the buffer count is decremented.

To encode non-deterministic flushes of the memory sub-system, a flush procedure is added by the translation function to the output program. The role of the flush procedure is to soundly encode the possible non-deterministic flushes of the store buffer, triggered by the memory subsystem. Just like the TSO case, for PSO a naive placement can be optimized using a simple preliminary static analysis that finds cases where the store buffer is guaranteed to be empty (and thus no flush is needed), or guaranteed to be bounded by a fixed size (and thus the flush code can be simplified).

## 5.2.5 Shortcomings of the Direct Translation

The main problem with the direct translation is that it performs operations that have a devastating effect on verification. Specifically for TSO: (i) the flush operation performs a shift of the array content, an operation that is very costly and makes it harder to track the relationships between values; (ii) the sizes of store buffers are tracked via numerical variables (i.e., cnt$_t$), the values of which may be lost under abstraction. As we show in Section 5.4, these shortcomings cause verification using direct translation to fail in more than 50% of our benchmarks, and to be costly for the remaining ones. In the next section, we present an abstraction and a translation which address these two shortcomings.

## 5.3 Abstraction-Guided Translation

We next present our new translation, which is based on a novel abstraction of the store buffers. We also contrast our approach with the direct encoding discussed earlier:

$$T_V : (Prog \times Rmm \times \mathbb{N}) \rightarrow Prog.$$
Our presentation focuses on the x86 TSO memory model (the details for PSO are similar). We first discuss the new abstraction, which eliminates shifting of values in the store buffers. Here, when an element is flushed from the buffer, the other elements maintain their position, significantly reducing the cost of the flush operation. This abstraction is generally applicable for any analysis. We then discuss an approach for replacing the counter variables that track the current size of the write buffers with boolean variables, which also improves precision when using abstract interpretation based analysis.

5.3.1 Robust Buffer Abstraction – Eliminating Buffer Shifting

The flush procedure appears at multiple places in the resulting program and hence its operation is critical to the overall precision and scalability of the analysis. As discussed earlier, the direct translation encodes a store buffer using two bounded arrays per thread (i.e. lhs and rhs) and a counter. If the bound is reached during analysis, an overflow error is triggered and the analysis aborts. When this happens, the user may increase the buffer bound, transform the program using the new bound, and rerun the analysis on the newly obtained program. The flush routine in the direct translation is implemented using a non-deterministic loop. In the loop body, the first element in the store buffer (the oldest) is flushed to memory. Next, the remaining elements are shifted one position to the left in the buffer. An advantage of shifting is that it frees entries at the end of the arrays encoding the buffer, thus creating free space for buffering additional store operations.

Key Idea: Our observation is that we can handle the flush operation without shifting the array content, thus obtaining an abstraction (over-approximation) of the relaxed memory semantics. This approximation is sound with respect to the direct translation (the proof is presented in Section 5.3.4) but may introduce additional cases of overflow. That is, if a program reaches an overflow when analyzed with our abstraction, it is possible that this overflow may not occur when using the direct, shifting encoding. However, we believe such situations are very rare in practice – in our evaluation in Section 5.4, no additional such overflows appeared in any of the benchmarks. We formally discuss how our abstraction is incorporated into the translation in Section 5.3.3.

5.3.2 Replacing Counters with Boolean Flags

Another ingredient of our approach is leveraging properties of the underlying program analysis. Unlike the general abstraction above, here we discuss an optimization suitable for abstract interpretation based analysis with numerical domains.

The direct translation keeps designated counters to track the current position in store buffers. When using numerical abstract domains such as Octagon [Min06] and Polyhedra [CH78], the exact numerical value of a variable may be abstracted away at program join points. This abstraction, desirable in most cases, has negative effects when
applied to key variables such as buffer counters. We would therefore like to keep the values of buffer counters even when different values for the count reach program join points.

Towards this, we use a logico-numerical domain, which combines a numerical domain and a logical domain that tracks boolean combinations of predicates. Rather than storing values of buffer counters as integers in the numerical part of the domain, we encode them using boolean variables in the logical part. This allows us to naturally maintain a disjunction of possible values for counters, without joining them. Using boolean variables to track buffer sizes therefore improves the precision of the analysis inside the flush procedure by differentiating cases where values of counter variables differ. This encoding can be viewed as a form of trace partitioning [RM07], where joins are delayed based on certain key values (in our case, the values of counter variables).

### 5.3.3 New Translation Rules for TSO

The source-to-source translation presented next incorporates both of the ideas described above. It replaces \( cnt \) counter variables with boolean variables. For each shared variable \( X \in Shared \), write buffer index \( i \in [1,b] \), and thread identifier \( t \in Thread \), a boolean variable \( flag_{X,i,t} \) is added.

If \( flag_{X,i,t} \) is true, then there is a shared variable \( X \) write in the thread \( t \) write buffer, to position \( i \).

The x86 TSO memory model has a single write buffer per thread. This translates to the invariant: for a fixed \( i \in [1,b] \) and a fixed \( t \in Thread \), there exists at most one shared variable \( X \) such that \( flag_{X,i,t} \) is true. In other words, at each location of the TSO buffer there is at most one shared variable write. We define the function:

\[
\text{OR}(i, t) = \bigvee_{X \in Shared} flag_{X,i,t}.
\]

The function \( \text{OR}(i, t) \) returns true if there exists a write (to any shared variable) at
the position \( i \) in the write buffer of thread \( t \). The previously mentioned invariant means that at most one disjunct will be true in the formula above. Fig. 5.6 shows the rules of the abstraction-guided translation:

**Write to a global variable** \([X = r]_b^t\): first checks if there is a write in the last element of the store buffer. If so, the analysis indicates an overflow and stops. If the store buffer is not yet full, the translation determines the highest index \( i \) in the buffer which is already occupied and places the current write at the position \( i + 1 \). Note that in each branch of the if-then-else statement, a boolean variable is modified. This enables the robust buffer abstraction (Sec. 5.3.1) and the boolean encoding of counters (Sec. 5.3.2).

**Read from a global variable** \([r = X]_b^t\): searches in the store buffer for the most recent write to the shared variable \( X \) and returns that value. If there is no write to \( X \) in the store buffer, then the value is read from the shared memory.

**Fence statement** \([\text{fence}]_b^t\): assumes that at this point the store buffer is empty – there are no pending writes.

**Flush action** \([\text{flush}]_b^t\): searches for the least recent entry in the store buffer and writes it to the shared memory. As opposed to the direct encoding, the element at position 1 is not flushed because the shifting procedure was removed. To know which variable is the buffered write, case testing is performed.

The new translation extends naturally to a sequence of statements and to programs with \( n \) concurrent threads: \([P]_b = [S]_b^1 \parallel \cdots \parallel [S]_b^n\).

### 5.3.4 Soundness of the Robust Buffer Abstraction

We next prove that the RBA abstraction incorporated in the translation \( T_V \) is sound as it over-approximates the direct translation \( T_D \). Given a program \( P \), memory model \( M \), and buffer bound \( b \), \( P^D = T_D(P, M, b) \) is the program that results from applying direct translation, and \( P^V = T_V(P, M, b) \) is the result of the abstraction-guided translation.

Fig. 5.8 summarizes the data structures needed to encode the write buffer of a thread \( t \) in the direct and abstraction-guided translations:

- \( B^D_t \) is the tuple containing the value of \( cnt_t \in [b] \) and the sequence of pairs of values for \( lhs_t \in \text{Shared} \) and \( rhs_t \in \mathbb{N} \), \( i \in \{1 \ldots b\} \).

- \( B^V_t \) is a sequence of elements which, for each shared variable \( X \in \text{Shared} \), associate a tuple containing the boolean variable \( \text{flag}_X^t \) and the stored value \( X^t \in \mathbb{N} \).
Let \( B^D = \{ B^D_t \mid t \in \text{Threads} \} \) and \( B^V = \{ B^V_t \mid t \in \text{Threads} \} \) be the sets of values of all write buffers of the programs \( P^D \) and \( P^V \).

We define the state of a translated program as the values of the shared variables, local variables, program counter, and auxiliary variables added by the translation: \( \sigma = (\text{Shared}_\sigma, \text{Local}_\sigma, pc_\sigma, B) \) or \( \sigma = \text{overflow} \). \( B \) is either the direct translation buffer state \( B^D \) or the abstraction-guided translation buffer state \( B^V \).

**Observable part of a state 1.** The observable part of a state includes: (i) the values of the shared variables, (ii) the values of the local variables, and (iii) the values and order between elements of the non-empty section of the buffer.

For \( T^D \), the observable part of the state contains the values of the shared and local variables and the values of \( \text{lhs}_i \) and \( \text{rhs}_i \) for \( i \in \{1 \ldots \text{cnt}_t\} \). Similarly, for \( T^V \), the observable part of the state contains the values of the shared and local variables and the values of \( x_i \) for \( i \) and \( t \), where \( \text{flags}_x \) is T.

**Equivalent states 2.** Two states \( \sigma^D \) and \( \sigma^V \) are equivalent if their observable parts correspond. (The global and local variables have the same values and the buffers \( B^D \) and \( B^V \) denote the same buffer content.)

We define the transitions between two states \( (\sigma_i, \sigma_{i+1}) \) for transformed programs as the translation rule (Fig. 5.4 or Fig. 5.6) corresponding to the transition in the original program \( P \). A trace of a program is represented as a sequence of states \( \pi = \sigma_1 \ldots \sigma_n \).

**Theorem 5.3.1.** The RBA abstraction used in \( T^V \) is sound. For any trace \( \pi^D = \sigma^D_1 \ldots \sigma^D_s \) of \( P^D \) of finite length \( s \), there exists a corresponding trace \( \pi^V = \sigma^V_1 \ldots \sigma^V_s \) of \( T^V \), such that for all \( i \in \{1 \ldots s\} \), \( \sigma^V_i \) and \( \sigma^D_i \) are equivalent or \( \sigma^V_i \) is overflow.

**Proof:** The proof is by induction on the length of \( \pi^D \).

First, we show how to build the trace \( \pi^V \). Given \( \pi^D = \sigma^D_1 \ldots \sigma^D_s \), the transition \( (\sigma^D_i, \sigma^D_{i+1}) \) for \( i \in \{1 \ldots s-1\} \) is a rule in Fig. 5.4, corresponding to the translation of an instruction in program \( P \). We construct \( \pi^V \) by applying at each step \( (\sigma^V_i, \sigma^V_{i+1}) \) the corresponding rule from Fig. 5.6.

Next, we prove that \( \pi^V \) and \( \pi^D \) have equivalent states.

Base case: for \( i = 1 \), in the initial state, all write buffers are empty, the shared variables have their initial values, and the local variables are not yet declared. Thus, states \( \sigma^V_1 \) and \( \sigma^D_1 \) are equivalent.

Induction step: for \( i > 1 \), we assume that the states \( \sigma^V_i \) and \( \sigma^D_i \) are equivalent or \( \sigma^V_i \) is overflow. If \( \sigma^V_i \) is overflow, then \( \sigma^V_{i+1} \) is also overflow (by convention, an overflow state cannot be changed).

If \( \sigma^V_i \) is not overflow, then the states \( \sigma^V_i \) and \( \sigma^D_i \) are equivalent (by the induction assumption). Our construction applies the transition \( (\sigma^D_i, \sigma^D_{i+1}) \) as defined by the rules in Fig. 5.4 and the corresponding transition \( (\sigma^V_i, \sigma^V_{i+1}) \) as defined by Fig. 5.6. We now
show that $\sigma_{i+1}^D$ and $\sigma_{i+1}^V$ are equivalent or $\sigma_{i+1}^V$ is overflow via case splitting on the transition type:

- **store**: write to a global variable $[X = r]_b$. Here, the local and shared variables remain unchanged. From the induction assumption $\sigma_i^D$ and $\sigma_i^V$, buffers hold the same values in the same order. From the assumption $\sigma_i^D \neq \text{overflow}$ and from the definition of store, we have that buffer content in $\sigma_i^D$ and $\sigma_i^V$ is the same or $\sigma_{i+1}^V$ will reach overflow.

- **load**: read from a global variable $[r = X]_b$. Here, the buffer contents are unchanged. The shared variables are also unchanged. From the induction assumption and the definition of load we have that the values of $r$ for $\sigma_{i+1}^D$ and for $\sigma_{i+1}^V$ are the same.

- **fence**: fence statement $[\text{fence}]_b$. Here, the transition assumes that at this point the store buffers are empty for both translations. The states do not change and the assumption on $\sigma_i^D$ and $\sigma_i^V$ propagates to the states $\sigma_{i+1}^D$ and $\sigma_{i+1}^V$.

- **flush**: flush action $[\text{flush}]_b$. Here, the local variables are unchanged. From the induction assumption, the buffers of $\sigma_i^D$ and $\sigma_i^V$ hold the same values in the same order, i.e., the same least recent element in the buffer will be flushed to main memory for $\sigma_{i+1}^D$ and $\sigma_{i+1}^V$.

This concludes the proof of Theorem 5.3.1 that $T_V$ is an over-approximation of $T_D$ and the RBA abstraction is sound. This also means that even if the trace $\pi^D$ does not end with an overflow, the corresponding trace $\pi^V$ might still do so.

**Circular buffers** An alternative to encoding the buffers into the program is to use a circular array. This approach requires auxiliary variables to keep track of the head and tail of the array. Each store operation adds an element at the tail of the array and each flush may remove one or more elements from the head of the circular array.

Circular buffers can provide precision similar to that of buffer shifting. However, we did not take this approach since it requires a more elaborate implementation, requiring awareness of when the buffer is circulated (half at the end of the physical implementation and half at the start). Additionally, more variables are required to keep track of the head and tail of the buffer, resulting in more overhead on the underlining analysis tool. Therefore we want to minimize the number of variables as much as possible by choosing the simple array encoding.

### 5.4 Evaluation

We implemented our approach and evaluated it on a range of challenging concurrent algorithms. We then compared its performance with the direct transformation discussed in Chapter 4. All our experiments ran on an Intel(R) Xeon(R) 2.13 GHz server with 250 GB RAM. To perform the analysis, we used ConcurInterproc [Jea13], a tool based on the APRON library [JM09], which supports various numerical abstract domains. We relied on the Z3 [DMB08] SMT solver to check that the inferred invariants imply the
Table 5.1: Verification results comparing our new transformation with the one in Chapter 4

<table>
<thead>
<tr>
<th>Program(Lines)</th>
<th>Model</th>
<th>Fences</th>
<th>b</th>
<th>Abstraction-guided translation</th>
<th>Direct translation</th>
<th>Chapter 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lines (sec) Time (MB) Lines Time (MB)</td>
<td>Lines Time (MB) Memory (MB)</td>
<td></td>
</tr>
<tr>
<td>Abp(60)</td>
<td>TSO</td>
<td>0</td>
<td>1</td>
<td>194 5 189</td>
<td>228 14 352</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>0</td>
<td>1</td>
<td>167 6 231</td>
<td>202 12 222</td>
<td></td>
</tr>
<tr>
<td>Bakery(88)</td>
<td>TSO</td>
<td>4</td>
<td>2</td>
<td>286 1148 4749</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>3</td>
<td>427 3429 10951</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Conloop(40)</td>
<td>TSO</td>
<td>2</td>
<td>2</td>
<td>260 8 547</td>
<td>276 18 891</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>2</td>
<td>2</td>
<td>272 6 504</td>
<td>324 23 783</td>
<td></td>
</tr>
<tr>
<td>Dekker(91)</td>
<td>TSO</td>
<td>6</td>
<td>2</td>
<td>494 227 2233</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>4</td>
<td>542 121 1580</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Kessel(44)</td>
<td>TSO</td>
<td>4</td>
<td>2</td>
<td>210 14 357</td>
<td>250 15 424</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>2</td>
<td>172 6 198</td>
<td>253 80 628</td>
<td></td>
</tr>
<tr>
<td>Loop2 TLM(65)</td>
<td>TSO</td>
<td>2</td>
<td>2</td>
<td>385 66 2234</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>2</td>
<td>1</td>
<td>302 36 1650</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Peterson(63)</td>
<td>TSO</td>
<td>2</td>
<td>3</td>
<td>379 89 1549</td>
<td>403 331 2280</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>2</td>
<td>275 20 901</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Psql(79)</td>
<td>TSO</td>
<td>3</td>
<td>2</td>
<td>567 282 1727</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>1</td>
<td>1</td>
<td>606 55 758</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Queue(41)</td>
<td>TSO</td>
<td>1</td>
<td>1</td>
<td>67 1 101</td>
<td>81 1 115</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>1</td>
<td>1</td>
<td>68 1 108</td>
<td>84 1 106</td>
<td></td>
</tr>
<tr>
<td>Sober(84)</td>
<td>TSO</td>
<td>2</td>
<td>4</td>
<td>647 30 1784</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>3</td>
<td>1</td>
<td>345 148 263</td>
<td>444 215 3499</td>
<td></td>
</tr>
<tr>
<td>Szymanski(110)</td>
<td>TSO</td>
<td>3</td>
<td>8</td>
<td>932 1066 3781</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>5</td>
<td>877 507 2076</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Chase-Lev WSQ(96)</td>
<td>TSO</td>
<td>2</td>
<td>2</td>
<td>284 17 550</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>1</td>
<td>192 9 520</td>
<td>229 10 528</td>
<td></td>
</tr>
<tr>
<td>THE WSQ(115)</td>
<td>TSO</td>
<td>4</td>
<td>3</td>
<td>460 125 1646</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSO</td>
<td>4</td>
<td>4</td>
<td>744 391 2338</td>
<td>- - -</td>
<td></td>
</tr>
</tbody>
</table>

specification.

The verification procedure has three steps:

(i) Applying the transformation on program $P$, obtaining a new program $P_M$.

(ii) Running ConcurInterproc on that transformed program.

(iii) Using Z3 to check whether the inferred invariants satisfy the specification.

The above procedure is repeated until it is no longer possible to further reduce the number of fences in the algorithm. We evaluated our approach on 13 concurrent algorithms, out of which 5 are infinite-state. The safety specifications are either mutual exclusion or reachability invariants involving labels of different threads.

Our main goal was to study the abstraction-guided translation precision and efficiency gains (i.e., memory consumption, speed) compared to the direct translation discussed in Chapter 4, while using the same analysis tool (in this case ConcurInterproc) to verify the output programs. Where applicable, we also discuss how our work compares to states of the art (here and in Section 5.5). Table 5.1 summarizes our experimental results for both the x86 TSO and PSO memory models.
The minimal number of fences necessary to verify each algorithm is shown in column 3 of Table 5.1. The used buffer size limits, referred to as b in Section 5.3, are shown in column 4. The number of code lines in the generated program, and time and memory resources used by the analysis, are shown for both the new transformation (in columns 5, 6, and 7 of the table) and the previous transformation (in columns 8, 9, and 10). We observe two trends:

- For Bakery, Dekker, Loop2-TLM, Pqsql, Szymanski and THE WSQ, the new transformation verifies the program with strictly fewer fences than the direct translation. The dash indicates that the verification failed for those placements, or their subsets, using the direct translation. (Failure in those cases was due to either out of memory or timeout.)

- For the rest of the benchmarks, the direct translation was successful in verifying the same fence placement as our new translation. But in all those cases:(i) the program generated by the new transformation had strictly fewer lines of code than the direct translation, and (ii) the time and memory consumption were better using the new translation, and in some instances (e.g., Sober) memory consumption was reduced by 10x.

We see that, for all the examples, as the buffer size grows so does the program size generated by the new translation. For example, for Szymansky, to generate a TSO program with 3 fences, we needed a minimal buffer size limit of 8 (this limit was found by the buffer analysis step); accordingly, the growth in program size was almost 9-fold. This was expected since the larger the buffer size limit, the more possible relaxed behaviors the new translation needs to account for. But, in all cases, the new translation generated strictly smaller programs than the direct translation.

We did not observe additional overflows introduced by the new translation that were not observed by the direct translation. This can be attributed to the buffer analysis generating buffer size limits, and the type of algorithms. We did generate a litmus test where such behavior was observed. However, since we did not find it in algorithms commonly used in our field, we believe our fabricated litmus test behavior is not common.

The addition of booleans to the program imposes an execution overhead on Concur-Interproc since its abstract domain keeps boolean variables concrete. The question of how to further minimize the number of boolean variables will be addressed in future work.

**Loops with at least one fence** Having at least one fence instruction in each loop improves the precision of the buffer size analysis because the bounds inferred by this analysis will not be infinite. However, our approach is not limited to programs with fences in each loop. For example, the ABP (Alternating Bit Protocol) benchmark is
verified by our tool with no fences added (each thread contains a loop). For ABP, despite the buffer analysis being imprecise, it is sufficient to set the maximum buffer bound $b$ to 1 and the verification succeeds.

**Choosing the buffer analysis constant** The buffer analysis receives as input a constant, which serves as a coarse guess for the maximal buffer size. During minimal fence placement search, we set this constant to be the number of removed fences in a fence placement we are trying to verify. In our search, full placement is a fence after each write. So, the number of removed fences is a bound on consecutive unfenced writes. If that bound is breached, by the buffer analysis, we have an unbounded potential buffer size. This bound breach might be caused by a loop with a write and no fence. A tighter bound can also be the maximal number of removed fences per process. The buffer analysis uses data flow to tighten the inputted constant to a smaller bound on the size of the buffer.

**Comparison to other work** Recent work [AKNP14] infers fences such that the program under the relaxed model is equivalent to SC – recall that we discussed such approaches as one of two general approaches in Section 5.2. Although scalable, the authors’ abstraction tends to lead to significant precision loss, thus inserting redundant fences. For instance, in Lamport’s Bakery under TSO, their abstraction inserts 8 fences, compared to 4 fences inserted by our analysis. This precision loss is observed also for other mutual exclusion algorithms such as Peterson under TSO (3 vs. 2 fences) and Szymansky under TSO (8 vs. 3 fences).

Another recent work [AAP15] infers fences such that the program is persistent (which is a concept defined in [AAP15]). The number of fences inferred for persistence is higher for some algorithms, although the approach is more scalable: for instance for Szymanski, under TSO [AAP15] infers 8 fences where our approach verifies Szymanski with 3. Comparison is not possible for other algorithms because [AAP15] infers fences with more than 2 processes.

Another line of work [AKNT13] also produces an SC program from the original program and the Relaxed Memory Model semantics. The work uses testing to find bugs in many litmus tests and algorithms (e.g., Bakery, Peterson, Dekker, Szymanski), but does not actually perform verification on any of them. Nor does it address the problem of how the proposed translation would affect infinite-state verification. For instance, when we tried [AKNT13] even on a few small examples, the resulting SC program used many more auxiliary boolean variables than our translation (e.g., 40 vs. 8). Note that even a small increase in the number of boolean variables quickly leads to state explosion (more disjunctions) in the program analysis. This is also confirmed by our experiments with ConcurInterproc, where, for instance, any program with more than 40 boolean variables could not be verified due to state explosion.

**Summary of Results** In summary, for each program, our new transformation enables
verification with a lower or equal number of fences compared to the direct translation. The new transformation also leads to a more efficient (in space and time) subsequent analysis of the resulting program. These results support our belief that our new abstraction-guided transformation is a key building block in automating verification of both finite and infinite-state concurrent programs on Relaxed Memory Models.

5.5 Related Work

We next discuss some additional work most closely related to ours. Over the last few years, there has been significant interest in ensuring correctness (via synthesis and verification) of concurrent programs running on Relaxed Memory Models. Most of the research has so far considered only finite-state programs or programs with unbounded store buffers, but where all shared variables range over finite domains [BDM13, BM08, BSS11, KVY10, KVY11, LW10]. Some recent works also handle infinite-state programs [AAC+12, AKNP14, DMVY13, SVN+13].

Previous works took the same approach as ours to handling Relaxed Memory Models: to encode the effect of the model directly into the program and then analyze the resulting program using tools that work for sequential consistency (e.g., in [AKNT13, ABP11, DMVY13]). The main contributions of our work are a new memory model abstraction and a transformation, which improve the precision and efficiency of the resulting program analysis. For instance, as we showed in the paper, using the direct encoding as in Chapter 4 will result in significant loss of precision and efficiency (i.e., failure to verify correct programs). Abdulla et al. [AAC+12] explore online predicate abstraction for handling infinite-state verification while Dan et al. [DMVY13] also explore predicate abstraction but this time based on offline analysis of boolean programs. Technically, these works are quite different from ours since: (i) they both use direct encoding and (ii) they both use predicate abstraction which, even with abstraction refinement, tends to require manually supplied predicates. In contrast, we provide a new robust abstraction of the store buffers and explore the application of numerical abstract domains that do not require manual annotations. We also provide a more comprehensive experimental study than either of the aforementioned works. (We consider x86 TSO, as well as the more relaxed PSO model and a range of challenging concurrent algorithms.)

For the common benchmarks, [AAC+12] and our approach achieve comparable results. A possible limitation of their work is locked writes, meaning that fences are generated immediately following a write to shared memory. Our tool is more flexible since fences can be placed at any label. We again note that the robust buffer abstraction (RBA) proposed in this work can be immediately useful with predicate abstraction as well. In [SVN+13], arbitrary safety properties are not taken into account. The technique supports two fence removal optimizations (for TSO), which are not enough to eliminate redundant fences. We applied the authors’ optimizations on a few of our benchmarks and, unfortunately, they failed to remove redundant fences (e.g., in the Chase WSQ...
Recently, the authors of [BCDM15] introduced a lazy approach to TSO state reachability. TSO reachability is reduced at each iteration of the lazy algorithm to SC reachability of a certain generated program. To determine the next iteration, a robustness checker is used. We believe that this lazy approach to reachability can be combined with our abstraction technique such that it enables studying reachability in infinite-state programs. In [AAP15] the authors present a new correctness notion called persistence and a pattern to check whether programs running on TSO are persistent (improving efficiency of checking robustness [BDM13]). The notion of persistence is more restrictive than reachability; thus, fences inferred for persistence may be redundant in contrast to fences inferred by our approach.

5.6 Conclusion

We proposed a new approach for verifying concurrent programs on Relaxed Memory Models. Our approach consists of a robust abstraction of the store buffers, an encoding of the store buffer sizes that leverages the underlying abstract domains, and a source-to-source translation that encodes Relaxed Memory Model semantics into the target program, thereby enabling the use of existing verification tools for sequential consistency.

We implemented our approach and evaluated it on a set of finite and infinite-state concurrent algorithms using an existing state-of-the-art abstract interpretation engine. Our experimental results demonstrate that the overall system is superior to prior work in terms of precision and performance, enabling verification of concurrent algorithms on both x86 TSO and PSO memory models not possible before.
Chapter 6

Pattern-based Synthesis of Synchronization for the C++ Memory Model

6.1 Introduction

With the goal of extending our work to other Relaxed Memory Models, we explored synthesis and verification under \textit{C++ Relaxed Memory Order} (C++ RMM) [ISO]. This memory model is more challenging than the previously described buffer based, hardware memory models (PSO, TSO). For example, it allows for future reads, which is another way of saying that the model allows concurrent programs to execute a load, reading from a store, where it is not possible to order, in time, the store before the load while still retaining consistency. Another challenge is that there is no known execution memory model for \textit{C++ Relaxed Memory Order}. This means that our previous, translation based, approach is not applicable to it.

Unlike our work for PSO and TSO, this work did not produce a method for verification of infinite systems under the \textit{Relaxed Memory Model} in question. Rather it is limited to programs with finite execution traces. Nonetheless we were able to describe a novel approach to synthesize program corrections.

The crucial task of writing correct and efficient low-level concurrent programs in C++ under this model is known to be very challenging: the model’s complexity is such that it eludes even veteran systems programmers and requires the attention of formal semantics experts [BOS+11, BDG13, VBC+15, TDV15].

Under C++ RMM, each operation on an atomic object is annotated with a \textit{memory order}. The memory order ranges from being fully \textit{relaxed} to being fully \textit{sequentially consistent} (for that atomic object), with a few more subtle modes between these two extremes. To maintain efficiency, the programmer wants the most relaxed synchronization required to preserve correctness, and nothing more (even when it simplifies
reasoning). Unfortunately, manually finding the right synchronization is extremely difficult, as it requires the programmer to reason about subtle interactions of the memory model. Our goal is to assist the programmer by automatically synthesizing the required synchronization.

6.1.1 The Problem

Given a finite-state program $P$ and a safety property $S$ such that $P \models S$ under a sequentially consistent (SC) memory model, we aim to automatically synthesize a program $P'$, whose behaviors are a subset of $P$’s behaviors, s.t. $P' \models S$ under C++ RMM in bounded executions.

6.1.2 Our Approach: Pattern Based Synthesis of Synchronization

Our synthesis algorithm automatically explores the (vast) space of programs that can be created from $P$ by modifying memory access synchronization. It does so by: (i) inspecting $P$’s (bounded) error traces to detect memory access patterns that can occur under C++ RMM but not under SC, and (ii) eliminating these error traces by preventing the occurrence of the detected violation patterns using as little synchronization as possible.

More specifically, our algorithm exhaustively explores the traces of $P$ under C++ RMM, and looks for error traces—traces which do not satisfy the specification $S$. If it finds an error trace, it searches it for instances of violation patterns, behaviors that may occur under C++ RMM but not under SC and that we know how to avoid. (Recall that $P$ satisfies $S$ under SC. Hence, violations of $S$ must be due to behaviors introduced by the weak memory model.) The algorithm then constructs a constraint which encodes all possible avoidance templates that can be used to eliminate that particular error trace. (Avoidance templates are strategies to synthesize memory order annotations of memory instructions such as load, store, and cas.) The algorithm accumulates the constraints required to eliminate the error traces and passes them to a SAT solver in the form of a CNF formula $\varphi$. Every satisfying assignment of $\varphi$ represents a different way to synthesize the desired memory order synchronization.

The algorithm then checks which of the resulting programs satisfies $S$. The check is required because our set of violation patterns and avoidance templates is not complete. (In fact, we believe that devising a complete set is nontrivial, if at all possible.) This means that a program $P'$ with no violation patterns may still violate the original specification $S$.

6.2 Overview

In this section, we provide an informal overview of our approach using our running example, Dekker’s mutual exclusion algorithm for two threads [Dij65].
Thread 0:  
1:  
2:  
3:  
4:  
5:  
6:  
7:  
8: ... // critical section 
9:  
10:  

Thread 1:  
1:  
2:  
3:  
4:  
5:  
6:  
7:  
8: ... // critical section 
9:  
10:  

Figure 6.1: Dekker’s mutual exclusion algorithm. Variables flag[0], flag[1] and turn are declared as atomic locations and initialized to 0. The subscripts indicate the synchronization (consistency) annotations synthesized by our tool.

6.2.1 Running Example

Fig. 6.1 shows one of the many variants of Dekker’s algorithm. The load (read) and store (write) commands are subscripted with memory order annotations. For now, these annotations can be ignored. The algorithm is comprised of: an entry section (lines 1–6) and an exit section (lines 9–10). The critical section itself (line 8) is irrelevant, and thus elided. The algorithm enforces mutual exclusion using variables flag[0] and flag[1], and ensures deadlock and starvation freedom using variable turn.

To enter the critical section, thread i, where i is either 0 or 1, needs to execute its entry section: First, it sets the value of variable flag[i] to 1 (line 1), thus signaling its intentions to the other thread. Then, it inspects the value of flag[1−i] to check whether the other thread is also trying to enter the critical section or is already in it (line 2). If not, it proceeds to the critical section. Otherwise, it sets its own flag to 0 (line 4), thus letting the other thread proceed, and waits for its turn to enter the critical section (line 5). Upon leaving the critical section, thread i executes the exit section, where it gallantly gives precedence to the other thread by setting turn to 1−i and signals that it left the critical section by setting the value of its flag to 0.

It is important to note that: (i) as long as a thread executes the critical section, its flag is set to 1; and (ii) a thread enters the critical section only after it ensures that the other thread’s flag is set to 0 while its own flag is set to 1. The above observation suffices to ensure mutual exclusion under SC, since, in this memory model, there is a total order between all the load and store commands and reading the value of a variable x returns the last value written to x. Thus, if two threads compete on entering the critical section, at least one must notice in line 2 that the flag of the other is set to 1.

Unfortunately, under C++ RMM this is no longer the case. The reason for this unintuitive behavior can be understood from the following simple program involving

\[\text{store}_{SC}(\text{flag}[0], 0);\]
\[\text{store}_{SC}(\text{flag}[1], 0);\]
\[\text{store}_{SC}(\text{turn}, 0);\]
only two store and two load commands.

**Example 6.2.1.** Consider the following program and assume that both \texttt{flag[0]} and \texttt{flag[1]} are initialized to 0, that \texttt{r[0]} and \texttt{r[1]} are initialized to 2, and that \texttt{r[0]} and \texttt{r[1]} are each local to the respective thread.

\[
\text{store}_W(\text{flag}[0], 1); r[0] = \text{load}_X(\text{flag}[1]) \parallel \\
\text{store}_Y(\text{flag}[1], 1); r[1] = \text{load}_Z(\text{flag}[0]).
\]

Under \textsc{SC}, at the end of the program the only possible values of \texttt{r[0]} and \texttt{r[1]} are 0 and 1. Furthermore, at most one of them can be 0. Under \textsc{C++ RMM}, \texttt{r[0]} and \texttt{r[1]} can be 0 simultaneously, for certain memory order annotations \texttt{W,X,Y}, and \texttt{Z}. This is because under \textsc{C++ RMM} a store operation can behave as if it writes its value to a thread-local store buffer, leaving the other threads to read the value stored in the global memory (\textsc{C++ RMM} exhibits x86-TSO behaviors).

The above example shows that mutual exclusion can only be ensured by adding synchronization to the program. One way to do it in \textsc{C++ RMM} is to explicitly annotate the load and store operations with the required synchronization type. Using strong synchronization primitives (e.g., requiring all load and store operations to be sequentially consistent) is expensive. Using synchronization primitives that are too weak, however, leads to unexpected behaviors. Thus, determining correct and efficient annotations is challenging. In contrast, our tool was able to determine that the program shown in Fig. 6.1 is safe if the memory operations in lines 1, 2, 3, and 9 are sequentially consistent, and the store in line 10 is memory order release\(^1\) (see Section 6.3).

\textbf{Note.} The load in line 5 is not synchronized (i.e., it is annotated with RLX). However, as we show in Section 6.5, our result is still verified by our underlying model checker.

### 6.2.2 Synthesizing Synchronization

Our approach rests on the insight that we can turn a program that is safe under \textsc{SC} into one that is safe under a weak memory model (\textsc{C++ RMM} in our case) by removing behaviors that cannot occur under \textsc{SC}. We face three main challenges in implementing this approach: (i) detecting such behaviors, (ii) determining a (cheap) way to remove them, and (iii) verifying that the resulting program is safe.

\textbf{Addressing the first challenge} We overcome the first challenge by exhaustively searching the program state space for an error trace, developing all the concrete traces possible under \textsc{C++ RMM}. We allow safety properties to be specified as: (a) assertions on the final state, (b) properties of thread-local variables, and (c) races on non-atomic locations (see Section 6.3). The search is guaranteed to terminate because we only follow bounded traces of finite state programs.

\(^1\)To the best of our knowledge, our solution is the only one to use memory order synchronizations and not fences.
**Addressing the second challenge** If we find an error trace, we look for instances of violation patterns, memory behaviors involving a small number of `load` and `store` actions possible under C++ RMM but not under SC and which we know how to prevent. Once we discover such an instance, we add synchronization annotations to the relevant memory operations using a predefined avoidance template that blocks the violation pattern, thus eliminating the error trace.

We describe the inferred synchronization annotations using a propositional formula and ask a SAT solver to find the sets of minimal satisfying assignments. (Note that a trace might contain several instances of violation patterns and thus can be eliminated using different avoidance patterns.) From each assignment, we generate a program and repeat the process until no bad trace is found. We use the verified solutions as a starting point in a new round of synthesis in which we raise the bound on the explored traces.

The algorithm is guaranteed to terminate because we consider only finite state programs, the number of memory annotations is finite, and every change only increases the degree of synchronization (see Section 6.3).

**Example 6.2.2.** Fig. 6.2 shows a trace of the Dekker algorithm that violates mutual exclusion. The trace contains two violation patterns, store buffering (SB) and load buffering (LB). The former, which we discussed in Example 6.2.1, is manifested here by the initialization `store` actions in lines 1 and 2, and the `load` actions in lines 5 and 8. (An `rf`-annotated arrow from a `store` action to a `load` action indicates that the latter read the value written by the former.) This instance of the SB pattern is blocked by synthesizing a SC annotation to the corresponding memory operations in the algorithms (lines i.1, i.2, 1, and 2 in Fig. 6.1).

*Note.* The list of violation patterns and their corresponding synchronization templates is given as an input to the algorithm. Our algorithm is parametric in that list. The specific patterns and templates that we use in our implementation are given in Section 6.4.2.

**Addressing the third challenge** Our set of violation patterns and avoidance templates is not complete. Thus, after synthesizing the programs, we simply explore the state space again. The synthesis procedure terminates if the offered solution contains only sequentially consistent memory accesses and is thus correct by our assumption, or when no error trace is found. This ensures that the program satisfies the desired properties in executions in which every thread performs no more instructions than the explored bound.

### 6.3 C++ Relaxed Memory Model in a Nutshell

A memory model defines the possible behaviors of instructions such as `load` and `store` in the program. Arguably, the most intuitive (and restrictive) memory model is **Sequential Consistency** (SC) [Lam79], in which there is a total order on the `load` and `store`
instructions, and every load from location \( l \) reads the last value stored in \( l \). (For simplicity, we treat the initial state as if it were produced by explicit store operations.)

The C++ Relaxed Memory Model is relational: (i) without relations no order of executing instructions is guaranteed; and (ii) a load can read from arbitrary stores. In addition, the model distinguishes between atomic locations, where racy accesses are allowed, and non-atomic locations, where the behavior of races is undefined. The locations we discuss next will be atomic. Below, we provide a (greatly simplified) overview of the part of C++ RMM relevant to our work.

We shall use Fig. 6.3 (SB) as a C++ Relaxed Memory execution trace example, though it was not intended as such and in Section 6.4 will be referenced in a different context. Assume a two-threaded program where: variables \( x \) and \( y \) are initialized to zero; one thread sets the value of \( x \) to 1 and another sets the value of \( y \) to 1; finally, each thread reads the variable set by the other thread.

The first relation we consider is read from (rf), denoted by \( \rightarrow_{rf} \), which relates store instructions to load instructions reading from them. The next relation we consider is happens before (hb), denoted by \( \rightarrow_{hb} \). For our purpose it is a transitively-closed union of the following relations (in general, in C++RMM, hb can be non-transitive): (i) sequence before (sb), denoted by \( \rightarrow_{sb} \), which places an irreflexive total order on the actions executed by the same thread; (ii) additionally synchronized with (asw), which relates instructions executed before thread creation to those executed by the thread, denoted by a dotted line separation; (iii) synchronized with (sw), which indicates instruction synchronizaon.

The model ensures that the only possible executions are ones in which these relations satisfy certain constraints. First, hb must be acyclic. Second, rf and hb should not

Figure 6.2: An error trace containing two violation patterns: (a) store buffering (SB) and (b) load buffering (LB). These patterns were detected by our tool when analyzing Dekker’s algorithm.
contradict each other: a load cannot read from a store that (i) depends on it, i.e., follows it in the \( \text{hb} \) relation, or (ii) is masked by another write, i.e., there exists a store operation such that \( \text{store} \rightarrow_{\text{hb}} \text{store}_2 \rightarrow_{\text{hb}} \text{load} \). Third, the \( \text{hb} \) induced instruction order should not contradict the modification order, which defines a total order on all store operations to the same location.

Note. Note that in Fig. 6.3 (SB) the aforementioned restrictions do not prevent reading values from initialization.

In addition, the \( \text{hb} \) relation should not contradict the memory order annotation. Every memory operation is annotated with a memory order annotation that specifies its consistency level: the level of synchronization and the ordering it requires. We consider three types of annotations:

(i) SC, whereby memory actions must be totally ordered;
(ii) Acq/Rel whereby a load_{Acq} that gets its value from a store_{Rel} imposes additional synchronization, and
(iii) Rlx, whereby operations do not place additional restrictions on the \( \text{hb} \) relation.

Note. For item (ii) above, these annotations induce a sw relation, and for item (i) sc (total order) relation is induced.

### 6.4 Synthesis of Synchronization

In this section we describe our synthesis algorithm (Section 6.4.1) and review the violation patterns and respective avoidance templates (Section 6.4.2) that we implemented and experimented with. We also present two abstract violation patterns that go beyond concrete litmus tests: we identify patterns involving a small number of memory operations on a single location, and describe how to block them by placing a chain of dependencies going through an unbounded number of accesses to (possibly) different locations (Section 6.4.3).

#### 6.4.1 Atomic Memory Access Synchronization Synthesis

Our synthesis procedure is comprised of two nested loops. The inner one synthesizes synchronization for a given program and the outer one keeps refining the set of solutions by gradually increasing the bound on the length of the explored traces.

Algorithm 6.1 implements the inner loop of the synthesis procedure. It takes as input a program \( P \) and a specification \( S \), and produces a set of programs \( P' \) which satisfy \( S \) under C++ RMM using different forms of synchronization.

The algorithm first checks whether \( P \) satisfies \( S \), and if so returns it (line 2). Otherwise, it goes over the set of traces which violate the specification (line 5) and looks for violation patterns in each trace. First, it searches for patterns which can be prevented using Acquire-Release synchronization (line 6); and only if no such patterns
Algorithm 6.1 The inner loop of the synthesis procedure.

1: Procedure $\text{SynSync}(P, S)$
2: if $P \models S$ then return \{P\}
3: $\phi = \text{true}$
4: $\mathcal{P} = \emptyset$
5: for $e \in \text{errorTraces}(P, S)$ do
6: $\beta = \text{blockOccurrence}(e, \text{AcqRelFix}(\cdot))$
7: if $\beta$ then continue
8: $\beta = \text{blockOccurrence}(e, \text{SCFix}(\cdot))$
9: if $\neg\beta$ then return allSC($P$)
10: $\phi = \phi \land \beta$
11: end for
12: $\phi = \phi \land \bigwedge \text{impliedSync}(\phi)$
13: avoidance = SAT($\phi$)
14: for annotation $\in$ avoidance do
15: $P' = \text{addSync}(P, \text{annotation})$
16: $\mathcal{P} = \mathcal{P} \cup \text{SynSync}(P', S)$
17: end for
18: return $\mathcal{P}$
19: Procedure $\text{blockOccurrence}(e, \text{patterns})$
20: $\beta = \text{false}$
21: for $(p, c) \in \text{patterns}$ do
22: for $i \in \text{occurrence}(p, e)$ do
23: $\beta = \beta \lor \text{blockPattern}(i, c)$
24: end for
25: end for
26: return $\beta$
27: end Procedure

28: $\text{impliedSync}(\phi) = \{a \rightarrow b \mid a, b \in \text{vars}(\phi) \land \exists \text{SC} \in \text{annot}(a) \land (\text{REL} \in \text{annot}(b) \lor \text{ACQ} \in \text{annot}(b)) \land \text{instr}(a) == \text{instr}(b)\}$
Algorithm 6.2 The synthesis procedure. Program $P$ is comprised of an initialization command $C_{\text{init}}$ followed by a parallel composition of $m + 1$ threads, where thread $i$ executes command $C_i$ for $N$ times.

1: Procedure $\text{PSynSync}(P, S, N)$
2: $C_{\text{init}}, C_0, \ldots, C_m = \text{getCmds}(P)$
3: $P_1 = \text{SynSync}(C_{\text{init}}; (C_0 \parallel \ldots \parallel C_m), S)$
4: for $n = 2$ To $N$ do
5: $P_n = \emptyset$
6: for $P' \in P_{n-1}$ do
7: $C_{\text{init}}, C_0, \ldots, C_m = \text{getCmds}(P')$
8: $\text{Loop}_0 = ' ' \text{for } i_0 = 1 \ldots n \text{ do } C_0,'$
9: $\text{Loop}_m = ' ' \text{for } i_m = 1 \ldots n \text{ do } C_m,'$
10: $P'' = ' ' C_{\text{init}}; (\text{Loop}_0 \parallel \ldots \parallel \text{Loop}_m)'$
11: $P_n = P_n \cup \text{SynSync}(P'', S)$
12: end for
13: end for
14: return $P_N$
15: end Procedure

are found in the trace does it search for patterns that can be prevented using the more expensive Sequential Consistency synchronization (line 8).

The search for instances of violation patterns and the corresponding avoidance template is done by the auxiliary procedure $\text{blockOccurence}(\cdot)$ (lines 19-27). If there is an instance $i$ of a pattern $p$ in trace $e$, then the avoidance template is instantiated according to the instance $i$ and recorded in $\beta$ as one way to eliminate trace $e$ (line 23). Technically, an instance of an avoidance-template is a conjunction of pairs $b = (\text{instr}(b), \text{annot}(b))$, (Each such pair is a boolean variable of the resulting formula) where $\text{instr}(b)$ is a load or a store in $P$ and $\text{annot}(b)$ is the suggested synchronization for that instruction: either SC, REL, or ACQ. The conjunction records the memory order annotations pertaining to the actions forming the detected instance $i$, which suffice to prevent it. Formula $\beta$ is constructed as a disjunction of ways to eliminate the trace $e$. The blocking formulae pertaining to all the error traces are accumulated as a conjunctive formula $\varphi$ (line 10).

Finally, we record in $\varphi$ that every constraint enforced by a REL or ACQ synchronization is also enforced by an SC synchronization by adding the corresponding implications (line 12, and explicitly given in line 28), thus increasing the set of possible solutions.

Every satisfying assignment to the program correction formula generates a different program, $P'$, which has more restrictive synchronization than $P$ (line 15). We determine whether $P'$ complies with the specification $S$, or requires further synchronization, by calling $\text{SynSync}()$ recursively.

If $\text{blockOccurence}()$ does not find a way to eliminate an error trace, we annotate all memory operations as SC (line 9).
Algorithm 6.2 implements the outer loop of the synthesis procedure. For simplicity, we assume that the input program is comprised of an initialization command $C_0$ followed by a parallel composition of $m + 1$ loops, where loop $i$ repeats executing a sequential command $C_i$ $N$ times.

The algorithm takes the original program $P$, a specification $S$, and the loop bound $N$, and generates a set of programs $P_N$ that restrict the synchronization in $P$ so that it satisfies $S$. Because the number of behaviors rapidly grows as loop iteration is increased, we take an incremental approach: we iteratively construct a sequence of sets of programs $P_n$, which satisfy the specification $S$ when each loop performs only $n$ iterations (lines 3 and 12). The programs in $P_n$ are used as a starting point in synthesizing programs with $n + 1$ iterations (line 7). Upon termination we return a set of different programs that refine $P$ using different memory order synchronization such that $P$ is compliant with $S$.

6.4.2 Patterns of Weak Memory Behavior

As mentioned previously, C++ RMM allows certain behaviors for a `load` that are not possible under SC. Below, we list some patterns of such behaviors and explain how they can be prevented using appropriate memory order annotations \cite{BDG13, BOS11}. The patterns can be seen in Fig. 6.3. We intuited the patterns from what are often referred to as litmus tests \cite{BOS11}.

**Store Buffering (SB)** This is the pattern from Fig. 6.2(a). In this pattern, two threads first write to two different locations and then try to determine the value of the location written by the other one. It is possible that each thread will not observe the `store` executed by the other. This behavior can occur when the stores of one thread are not immediately visible to the other.

*Pattern prevention.* This pattern can be prevented only by making all the `load` and `store` instructions SC.

**Independent Reads of Independent Writes (IRIW)** Here two threads write to two different locations and the other two threads see those writes in different orders.

*Pattern prevention.* The above pattern can be prevented only by making all the `load` and `store` instructions SC.

**Load Buffering (LB)** This is the pattern from Fig. 6.2(b). This pattern indicates that every thread can see later (according to the $sb$ relation) writes of the other threads. Note that as the `store` might actually be dependent on the `load`, this pattern indicates that each thread can “magically” satisfy the needs of the other. Hence, this pattern is also called satisfaction cycles or reading values out-of-thin-air.

*Pattern prevention.* Adding one of the `rf` edges to `hb` would prevent this pattern. This can be done by annotating the `store` and `load` instructions of that edge with REL and ACQ, respectively.
Figure 6.3: Patterns of behaviors possible under C++ RMM but not under SC. Every column depicts the actions of one thread. We denote by store $x$, 1 a write of value 1 to location $x$ and by load $x(1)$ a read of value 1 from $x$. We assume that the initial value of $x$ and $y$ is 0.

**Message Passing (MP)** Here, one thread writes to two different locations, and the other thread sees the value written by the second store (to $y$), but misses the first store (to $x$).

*Pattern prevention.* Annotating the store to $y$ with REL and the load from $y$ with ACQ would add the rf edge to the hb relation and prevent the pattern.

**Write-to-Read Causality (WRC)** This pattern is similar to the message passing pattern, but involves three threads. Here, the value written to $x$ by the first thread is read by the second thread, which then, according to the sb order, writes a value to $y$. The third thread sees the value written by the second thread but not by the first.

*Pattern prevention.* Annotating the load and store with REL and ACQ respectively would prevent this pattern.
6.4.3 Abstracting the Patterns

The presented pattern list captures several behaviors of C++ RMM. Instances of those patterns were observed in almost all of our benchmarks but there are still C++ RMM behaviors not captured by the previous list. What’s more, the patterns share some similarities. In an attempt to bring us closer to completeness, we drew on that resemblance and extracted the commonalities into abstract patterns.

Using the RD property in (RD₁, RD₂) The following patterns are motivated by the RD property defined in [BDG13]. The relation R can be instantiated in two different ways: first as a transitive closure of rf and hb relations, and second as a possible total order on the involved instructions.

For the first instantiation, the relation R is the transitive closure of rf ∪ hb. Making all load instructions Acq and all store instructions Rel across the path will add all the rf edges along the path in R to hb, forming a sequence violating the RD property in [BDG13] and preventing that behavior.

When we cannot find such instantiation of the relation R in the error trace, we try to instantiate it as a possible total order of instructions, and prevent the error trace using SC. In our implementation we chose to attempt instantiation of R as the scheduler choice made by CDSChecker. One such scheduling choice, exemplified in Fig. 6.2(a) as the index of instructions 1-15, is a possible total order which the SB pattern violates. Forcing total order of instructions involved in the pattern (making the memory order access SC) will cause the load to violate the RD property.

The following points should also be noted: 1. RD₁ with R as rf ∪ hb transitive closure is an abstraction of the message passing (MP) pattern. 2. RD₂ with R as rf ∪ hb transitive closure is an abstraction of the load buffering (LB) pattern. 3. RD₁ with R as a possible instruction total order is an abstraction of the store buffering (SB) pattern. 4. RD₂ with R as a possible instruction total order is a read from future C++ relaxed behavior.
Table 6.1: Results of synchronization synthesis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>N</th>
<th>time (s)</th>
<th>space (Mb)</th>
<th># calls</th>
<th>patterns observed</th>
<th># solutions</th>
<th># bad traces N=1</th>
<th>inferred synch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternating Bit Protocol (abp)</td>
<td>5</td>
<td>206.89</td>
<td>22</td>
<td>1</td>
<td>MP(SC), RD_1, RD_2</td>
<td>5</td>
<td>(N=1,2) 0, (N=3) 1</td>
<td>(5, 0, 0, 1)</td>
</tr>
<tr>
<td>dekker [Dij65]</td>
<td>1</td>
<td>3m:22</td>
<td>22</td>
<td>3</td>
<td>MP, LB, SB, RD_1, RD_2, RD_1(SC), RD_2(SC), RD_4</td>
<td>13</td>
<td>631</td>
<td>(10, 1, 0, 8)</td>
</tr>
<tr>
<td>d-prcu-v1 [AM15]</td>
<td>3</td>
<td>3m:14</td>
<td>19</td>
<td>20</td>
<td>LB, SB, RD_2, RD_1(SC), RD_2(SC),</td>
<td>7</td>
<td>5</td>
<td>(7, 2, 1, 0)</td>
</tr>
<tr>
<td>d-prcu-v2 [AM15]</td>
<td>3</td>
<td>3h:53m</td>
<td>22</td>
<td>88</td>
<td>MP, LB, RD_2, RD_1(SC), RD_2(SC),</td>
<td>17</td>
<td>8</td>
<td>(9, 2, 1, 4)</td>
</tr>
<tr>
<td>kessel [Kes82]</td>
<td>3</td>
<td>57m:16</td>
<td>22</td>
<td>5</td>
<td>MP, LB, SB, RD_1, RD_2, RD_1(SC), RD_2(SC)</td>
<td>2</td>
<td>85</td>
<td>(13, 1, 0, 0)</td>
</tr>
<tr>
<td>peterson [Pet81]</td>
<td>3</td>
<td>26m:41</td>
<td>22</td>
<td>3</td>
<td>MP, LB, RD_1, RD_2, RD_1(SC), RD_2(SC)</td>
<td>(N=1) 2*</td>
<td>37</td>
<td>(11, 1, 0, 0)</td>
</tr>
<tr>
<td>bakery [Lam74]</td>
<td>4</td>
<td>10m:21</td>
<td>33</td>
<td>3</td>
<td>MP, LB, RD_1, RD_2, RD_1(SC), RD_2(SC)</td>
<td>(N=1) 6</td>
<td>974</td>
<td>(16, 1, 0, 0)</td>
</tr>
<tr>
<td>ticket [And91]</td>
<td>4</td>
<td>1m:08</td>
<td>19</td>
<td>7</td>
<td>RD_1(SC), RD_2(SC)</td>
<td>4</td>
<td>8</td>
<td>(9, 0, 0, 1)</td>
</tr>
<tr>
<td>treiber stack [Tre86]</td>
<td>1</td>
<td>1h:05</td>
<td>23</td>
<td>1</td>
<td>MP</td>
<td>1</td>
<td>160</td>
<td>(0, 5, 3, 4)</td>
</tr>
</tbody>
</table>

6.5 Implementation and Experimental Evaluation

We implemented our approach in a tool called PSYN SYN, which is based on CDSChecker [ND13, ND16]. Our tool computes a symbolic formula that captures possible fixes, and uses Z3 to find minimal satisfying assignments. Then, we thoroughly evaluated the tool on a number of challenging concurrent algorithms. For all benchmarks our tool found a nontrivial solution with non-SC memory accesses. All experiments were conducted on an AMD Opteron Processor 6376 with 128GB RAM and 64 cores, but using only a single thread per benchmark execution. The synthesized solutions and visual tools that explain our work are available at [PSy].

The results of the experimental evaluation are summarized in Table 6.1. Most table columns are self-explanatory, but we elaborate on the following:

- The $N$ column shows the maximal number of iterations we attempted for each thread.
- The patterns observed column shows for each algorithm the instances of patterns described in Section 6.4.2.
- The $\#$ solutions column shows the number of solutions we found for the benchmark. Unless otherwise specified, the solutions are for maximal $N$ attempted.
- The $\#$ bad traces for $N=1$ column shows the number of bad traces CDSChecker found in the original benchmark with each process doing 1 iteration.
- The inferred synch column shows the number of memory access synchronizations of every type suggested by our tool in every solution. Due to space restrictions,
we present synchronization of up to 3 solutions per benchmark and use “…” if more solutions exist.

All our benchmarks, when providing an error trace, exhibited one of the patterns. For the RD_1 and RD_2 patterns, SC notation is used when the relation R was instantiated by a possible instruction scheduling and SC synchronization was required to prevent the error trace. In addition, the RD pattern occurrences are reported only if they could not be captured by patterns from Fig. 6.3. This is the case, for example, for pattern instances that are similar to MP but whose path from store x,1 to load x(0) involved more than three sb ∪ rf edges and so can only be classified as RD_1 and cannot be classified as MP.

For ABP we can see that the original algorithm was verified when each process performed 1 iteration. It was not until each process performed 3 iterations that a violation of the checked property was encountered. At that point 1 error trace was found but it exhibited several patterns; therefore several ways of preventing it were found. We found 5 solutions that verified for 3 iterations of ABP, and those solutions verified for 4 and 5 iterations as well.

In fact, for almost all our benchmarks, solutions, once found, remained verified solutions when more iterations per thread were attempted. This was not the case for Peterson, as indicated by the “*” in the last column. Here the solution (11, 1, 0, 1) (which are (SC, REL, ACQ, Rlx) respectively) found in 1 iteration had a mutual exclusion breach when attempted with 2 iterations, and the solution was further restricted by making the one relaxed memory access SC, thus turning it into the solution marked with “*”. That solution later verified for 3 iterations.

For Bakery, the 6 solutions in iteration 1 reduced to a subset of 4. For all other benchmarks the solutions in the last column were found with 1 iteration per process and remained verified for the maximal number of iterations attempted.

Previous attempts (e.g. [VBC+15]) were made to verify RCU under C++RMM, but the version we verified was the first one where an update waits only for the reads whose consistency it affects, and does not wait for the completion of all existing reads.

For Dekker’s algorithm, we are not aware of any previous attempt to synthesize the correct version of it using memory accesses instead of fences (one such fence solution is a benchmark of CDSChecker). The solution found by our tool seems more restrictive than the fence based solution: for example, in our solution, the load of flags at the while condition creates fences (when translated to intermediate code) at the exit and at the entry of the loop; in the fenced version, however, a fence appears only after the loop exit and not at the entrance. What’s more, where the fence placements do correlate, ours are still more restrictive, perhaps due to the incompleteness of our set of patterns and corrections.

For Treiber’s stack algorithm, CDSChecker had a synchronized verified version. For it, our proposed solution was more restrictive than the manual one provided by CDSChecker.
6.6 Related Work

In this section, we review some closely related work, including synthesis of synchronization, automatic verification, bounded model checking, and dynamic analysis.

**Fence synthesis for x86-TSO and PSO** Existing techniques for synthesizing synchronization for Relaxed Memory Models have focused on hardware memory models. Kuperstein et al. [KVY10] presented a framework for fence inference in hardware memory models such as PSO and TSO. Their framework is based on a simple operational semantics that explicitly tracks store buffers to capture effects of the Relaxed Memory Model. They later [KVY11] extended their technique using abstractions of unbounded store buffers. This allowed them to scale their technique and handle a larger set of algorithms. Abdulla et al. [AAC+12] infer memory fences for infinite-state programs under x86-TSO by combining predicate abstraction with abstractions of store buffers.

Our technique synthesizes synchronization for the C++ Relaxed Memory Model. We note that the memory behavior under TSO and PSO is captured by the SB violation pattern.

**Formalizing C++ RMM** Batty et al. [BOS+11, Bat14] formalized the C++ RMM and proved correctness of compilation onto TSO and Power [Pow]. These works inspired our definition of violation patterns and avoidance templates. We also intuited from the formal model when generalizing the concrete violation patterns into abstract ones. Their tool, CPPMEM, bears some similarity to CDSChecker, which we use in our implementation. Thus, we believe that it would be possible to incorporate CPPMEM into our synthesis procedure.

**Program logics for C++RMM** Vafeiadis et al. [VN13, TVD14] developed a Hoare-style program logic verification technique that extends separation logic [ORY01, VP07] to C++ RMM. Batty et al. [BDG13] provided an extension of linearizability and verified that an implementation of Treiber’s stack [Tre86] corresponds to an abstract stack under C++ RMM. These works allow for manual verification. Our synthesis procedure is based on Bounded Model Checking. However, if these works pan out to automatic verification techniques, it should be fairly straightforward to combine them with our technique as a final stage in which we verify the synthesized solutions.

**Fence synthesis for x86-TSO, PSO and IBM Power** C++ RMM was developed with underlying hardware in mind. The following works should therefore shed some light on the behaviors it allows. Joshi et al. [JK15] introduced Reorder Bounded Model Checking. Their approach is based on instruction reordering, and their tool synthesizes minimal fence placement. We, on the other hand, synthesize memory order synchronization. It would be interesting to see whether our technique can be combined with theirs. Musketeer, developed by Algave et al. [AKNP14], provides a flexible scheme for fence synthesis to ensure robustness, i.e., that every concurrent execution be observationally equivalent to a serial execution. CheckFence of Burckhardt
et al. [BAM07] also ensures robustness by converting a program into a form that can be checked against an axiomatic model specification. Our technique makes it possible to verify user-provided safety properties.
Chapter 7

Extrapolation

In the previous chapters we looked at various Relaxed Memory Models and, using knowledge about Sequential Consistent behaviors, verified and synthesized correct versions of the programs. In this chapter we would like to generalize this notion of using knowledge from a simpler domain to reason about the program under more relaxed domains.

Notations
A relation between two operational semantics $\llbracket_1$ and $\llbracket_2$ such that for any program $P$, $\llbracket P \rrbracket_2$ contains all the behaviors of $\llbracket P \rrbracket_1$ will be denoted by $\llbracket_1 \leq \llbracket_2$.

Assumptions
Given an abstraction scheme $\alpha$ and two semantics $\llbracket_1$ and $\llbracket_2$ such that $\llbracket_1 \leq \llbracket_2$, we assume:

1. Verification under $\llbracket_1$ and $\alpha$ is feasible - The set of program-specification pairs $A = \{(P, S) \mid \llbracket P \rrbracket_1 |\alpha S$ can be checked with reasonable resources$\}$ is not empty.
2. Verification under $\llbracket_2$ and $\alpha$ is not feasible - The set $A$ of Assumption 1 has pairs $(P, S)$ such that $\llbracket P \rrbracket_2 |\alpha S$ cannot be checked without reasonable resources.

We introduce the following hypothesis:

*Hypothesis 7.0.1.* Given two semantics $\llbracket_1 \leq \llbracket_2$, a program $P$ with a specification $S$ and an abstraction $\alpha$ such that $\llbracket P \rrbracket_1 | S$, $\llbracket P \rrbracket_2 | S$ and $\llbracket P \rrbracket_1 |\alpha S$ but $\llbracket P \rrbracket_2 \not{|\alpha S}$ for some reason in a straightforward attempt (though it does hold): the proof of $\llbracket P \rrbracket_1$ can be used to enable the successful proof of $\llbracket P \rrbracket_2 |\alpha S$.

7.1 Model Checking with Symmetry

Our search for a concrete example of geometric extrapolation led us to look at programs for GPU and specifically at the Parallel Programming and Computing Platform CUDA [CUD]. Fig. 7.1 is an example taken from a CUDA tutorial. In the example we have code for two executing CPU cores. The code for the two cores is identical except for
Initially: \( \forall i. (a[i] = -1 \land b[i] = 1) \).

We want to prove \( \forall i. c[i] = 0 \) after execution.

Figure 7.1: CUDA array summation example.

We assume array \( a \) is initialized to \(-1\) at every cell and array \( b \) is initialized to 1 at every cell. Semantically, the operations the two cores perform are symmetrical, the symmetry being in this case transposition. Each core sums the even (for \( tid \) initialized to 0) or odd (for \( tid \) initialized to 1) cells, of the same index, from arrays \( a \) and \( b \) into \( c \). If we prove that: (1) CPU core 1 sets all even cells of \( c \) to zero; (2) CPU core 2 code is a transposition of core 1 code; and (3) the code for core 1 and 2 do not interfere, we can use the proof of item 1 for the code of core 2 (after transposition) and arrive to the conclusion that all odd cells of \( c \) will be zero at the end of its execution. Since the two cores do not interfere with each other we can take the two conclusions and combine them in conjunction to get the final desired conclusion that \( c \) will be all zeroes at the end.

7.1.1 Proof of Concept

We will first produce a proof of \([P]_1\), which in our case is the code of the program under \( tid == 0 \). \([P]_2\) in this case will be the program for two processes run concurrently. Therefore, we have two stages of semantic complexity:

The first stage is taking the proof of the original program to its symmetric case for \( tid == 1 \). This will be done using a shifting of all the predicates, and extrapolating the states we receive for the proof of \([P]_1\).

The second stage is taking the proofs of the original and the symmetric case and combining them to a proof of the program. This will be done using the Owicki-Gries technique.

A note about notation: We will replace \( tid \) with \( tid0 \) in the instruction and predicates for core 1, and with \( tid1 \) in the instruction and predicates for core 2. In addition, we will use uppercase letters for the shared arrays \( A \), \( B \) and \( C \). For example, we will write ‘\( C[tid0] = A[tid0] + B[tid0] \)’ for the code of core 1 and ‘\( C[tid1] = A[tid1] + B[tid1] \)’ for the code of core 2.

To prove the correctness of the example of Fig. 7.1, we chose the set of predicates
listed in Fig. 7.2. Other choices were possible, perhaps smaller sets. Our choice of that particular set was motivated by the following assumptions:

1. The underlining predicate proof setting is translation to a boolean program and using a model checker. (Note: Here we imply three valued logic.)

2. The theorem prover theory is first order logic, with array cells treated as separate variables.

**Proof for tid == 0**

Fig. 7.3 presents the proof of the program in Fig. 7.1 for tid = 0. For brevity we write after each line only those predicates whose value changes. For unknown predicates we use the notation “∗”. For the undefined array C we use the notation “⊥”. The predicates are written as a conjunction inside parentheses symbolizing a boolean program state which is a vector of zero/one/asterisk.

In this example we delayed performing joins of states, in effect doing loop unrolling. This limits our analysis to finite state programs. Performing joins for the selected predicates, would have caused verification to fail. The joins, after lines 2 and 3, would have caused $P_8$ and $P_9$ to be unknown. A more robust approach would find a loop invariant capturing the relation between tid and arrays a,b, and c indexes, which might require a higher order logic. Our approach, is closer in spirit to translation to a boolean program and using state collecting semantics, allowing us to use simpler predicates.

The initialization is: (i) predicates that capture the tid value, $P_1 = (tid0 == 0)$, $P_2 = (tid0 == 2)$, and $P_3 = (tid0 == 4)$, are unknown; (ii) predicates that capture the values of arrays “A” and “B”, $P_4 = (A[0] == 1)$, $P_5 = (A[2] == 1)$, $P_6 = (B[0] == -1)$, and $P_7 = (B[2] == -1)$, are true, and will remain so throughout the program; (iii) predicates that capture the value of array “C”, $P_8 = (C[0] == 0)$ and $P_9 = (C[2] == 0)$, are false.

For correctness we want to prove that both $P_8$ and $P_9$, are true at the end of the execution.

The assignment of ‘tid0=0;’ sets predicate $P_1$ to be true and predicates $P_2$ and $P_3$ to be false. In the while loop we have two states representing two subsequent iterations of the loop. The left state is the first iteration (where $P_1$ is still true).

At the first iteration of the loop, since $P_1$ is true, we receive after line 3 that $P_8$ is true. Then, after line 4, we receive $P_2$ is true, and $P_1$ and $P_3$ are false. Finally, with $P_8$ true we proceed to the second iteration of the while.

At the second iteration of the loop, since $P_2$ is true we receive after line 3 that $P_9$ is true. Now, after line 4, $P_3$ is true and $P_1$, and $P_2$ are false. With both $P_8$ and $P_9$ true, we proceed to exit the while loop and successfully finish the program verification.
void add(int *A, int *B, int *C) {
    int tid0 = 0;
    {
        (P_1 \land \neg P_2 \land \neg P_3 \land \ldots)
    }
    while (tid0 < N) {
        {
            (P_1 \land \neg P_2 \land \neg P_3 \land \ldots), (\neg P_1 \land P_2 \land \neg P_3 \land \ldots)
        }
        C[tid0] = A[tid0] + B[tid0];
        {
            (\cdots \land P_8 \ldots), (\ldots \land P_9)
        }
        tid0 += 2;
        {
            (\neg P_1 \land \neg P_2 \land \neg P_3 \land \cdots \land P_8 \land P_9)
        }
    }
    {
        (\neg P_1 \land \neg P_2 \land P_3 \land \cdots \land P_8 \land P_9)
    }
    assert(P_8 \land P_9)
}

Figure 7.3: CUDA example proof for tid0=0

Proof for tid1 == 1

We begin with a mapping \( m : \text{Integer} \to \text{Integer} = \{ \text{tid} \mapsto \text{tid} + 1 \} \), which we will then extend.

First, extending \( m \) to predicates, for each predicate \( P_i \) (where \( i \in \{1, \ldots, 9\} \)) we receive a corresponding predicate \( P^m_i \). The predicates are listed in Fig. 7.4.

Next, we take the proof in Fig. 7.3, and at each point where we had \( P_i \) in the state, we write \( P^m_i \). Now, we obtain the proof of the example in Fig. 7.5. Note that generally, up to this point, this is a simple rewrite step resulting in a proof of the program for which we don’t need a model checker.

Finally, we need to do a single (linear) pass over the proof to make sure it is correct. That is, we need to check that the set of states before each instruction, which can be seen as a disjunctive normal form formula, implies, after the instruction’s application, the set of states following the instruction.

For line 3, for example, we need to check that \( \{ (P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots), (\neg P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots) \} \) after the application of ‘\( C[tid1] = A[tid1] + B[tid1] \);’ implies \( \{ (\cdots \land P^m_8 \land \neg P^m_9), (\cdots \land \neg P^m_8 \land P^m_9) \} \). This instruction effects only predicates capturing properties of the array “C”, which are \( P^m_8 = (C[1] == 0) \) and \( P^m_9 = (C[3] == 0) \).
\[ P^m_1 = (\text{tid}1 == 1) \quad P^m_4 = (A[1] == 1) \quad P^m_8 = (C[1] == 0) \]
\[ P^m_2 = (\text{tid}1 == 3) \quad P^m_5 = (A[3] == 1) \quad P^m_9 = (C[3] == 0) \]
\[ P^m_3 = (\text{tid}1 == 5) \quad P^m_6 = (B[1] == -1) \quad P^m_7 = (B[3] == -1) \]

Figure 7.4: CUDA example mirrored predicates

variable initialization:

subsequent initial predicate values:
\[ P^m_{1-3} = \ast; P^m_{4-7} = T; P^m_{8,9} = F \]

```c
void add( int *A, int *B, int *C){
  int tid1 =0;
  {((P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots \land \neg P^m_8 \land \neg P^m_9))}
  2: while (tid1 < N) {
    {((P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots \land \neg P^m_8 \land \neg P^m_9), (\neg P^m_2 \land P^m_2 \land \neg P^m_3 \land \ldots \land P^m_8 \land \neg P^m_9))}
    {((\ldots \land P^m_8 \land \neg P^m_9), (\ldots \land P^m_8 \land P^m_9))}
    4: tid1 += 2;
    {((\neg P^m_1 \land P^m_2 \land \neg P^m_3 \land \ldots), (\neg P^m_1 \land \neg P^m_2 \land P^m_3 \land \ldots))}
  5: assert(P^m_8 \land P^m_9)
}

Figure 7.5: CUDA example mirrored proof for tid1=1
```
will notate \( I = \text{C[tid1]} = A[tid1] + B[tid1] \); and then use Hoare’s triplets, logic notation, to signify implication after instruction’s application.

Since \( P^m_1 = (\text{tid1} == 1) \) after application of \( I \) implies \( P^m_8 \), which we notate \( \{P^m_1\}I\{P^m_8\} \), we have \( \{(P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots)\}I\{(P^m_8 \land \neg P^m_9)\} \) as well as \( \{(P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots)\}I\{(\cdots \land P^m_8 \land \neg P^m_9)\} \); the latter holds due to the predicates not changing during the application of \( I \). Next, we can use properties of implication to weaken the implied target and obtain:

1. \( \{(P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots)\}I\{(\cdots \land P^m_8 \land \neg P^m_9) \lor (\cdots \land \neg P^m_8 \land P^m_9)\} \).

In a similar manner, from \( P^m_1 = (\text{tid1} == 1) \) implying \( P^m_9 \) after application of \( I \), we will obtain that \( \{(-P^m_1 \land P^m_2 \land \neg P^m_3 \land \ldots)\}I\{(\cdots \land P^m_8 \land \neg P^m_9)\} \) holds and:

2. \( \{(-P^m_1 \land P^m_2 \land \neg P^m_3 \land \ldots)\}I\{(\cdots \land P^m_8 \land \neg P^m_9) \lor (\cdots \land \neg P^m_8 \land P^m_9)\} \).

From 1 and 2 we will receive the desired result:

\( \{(P^m_1 \land \neg P^m_2 \land \neg P^m_3 \land \ldots) \lor (\neg P^m_1 \land P^m_2 \land \neg P^m_3 \land \ldots)\}I\{(\cdots \land P^m_8 \land \neg P^m_9) \lor (\cdots \land \neg P^m_8 \land P^m_9)\} \).

Similar reasoning holds for all other implications in Fig. 7.5 and can be implemented in a single pass over the program using a theorem prover such as, e.g., Z3.

**Owicki-Gries**

Having two proofs, we can now create their conjunction. We will take each state over predicates \( P_1 - P_9 \) and extend it with possible states over \( P^m_1 - P^m_9 \), receiving a set of states where each contains over 18 predicates. We will have a cross-product of states, since when CPU core 1 of Fig. 7.1 is at some line, CPU core 2 can be at any one of its lines. What helps us in this case is that \text{tid} is local to each process and the two proofs are disjoint, that is, interference free.

Generally, given two statements \( S_1, S_2 \) and predicates \( Q_1, Q_2 \) that hold after them, that is \( \{\text{pre}(S_1)\}S_1\{Q_1\} \) and \( \{\text{pre}(S_2)\}S_2\{Q_2\} \), to prove interference freedom according to Owicki-Gries we need to prove:

1. \( \{\text{pre}(S_1) \land \text{pre}(S_2)\} S_1 \{\text{pre}(S_2)\} \)
2. \( \{\text{pre}(S_1) \land \text{pre}(S_2)\} S_2 \{\text{pre}(S_1)\} \)
3. \( \{Q_1 \land \text{pre}(S_2)\} S_2 \{Q_1\} \)
4. \( \{\text{pre}(S_1) \land Q_2\} S_1 \{Q_2\} \)

We can choose as \( S_1 \) and \( S_2 \) any parts of program cores 1 and 2 or the two programs as a whole. The interesting lines are lines 3 from the code of each core, since they touch shared variables – the cells of the array “C”. We will exemplify the process next to lines
3 of both core 1 and core 2. We will write $S_1 = 'C[tid0] = A[tid0] + B[tid0]';$ for the code of core 1 and $S_2 = 'C[tid1] = A[tid1] + B[tid1]';$ for the code of core 2.

In $\{pre(S_1) \land pre(S_2)\}$ $S_1 \{pre(S_2)\}$, the instruction $S_1$ is over $tid0$, and $tid0$ is either 0 if $P_1$ is true, or 2 if $P_2$ is true. Therefore, in $S_1$ either the value of $C[0]$ or the value of $C[2]$ is changed. None of the values of $P_m^{m} - P_m^{0}$ is affected by it, and $\{pre(S_2)\}$ is not affected by $S_1$, so we get that $\{pre(S_1) \land pre(S_2)\}$ $S_1 \{pre(S_2)\}$ holds as desired.

By similar reasoning, 2-4 hold, and we have interference freedom. Extending this to the entire program, we conclude that $P_8 \land P_9$ conjuncted with $P_8^{m} \land P_9^{m}$ hold at the end of execution, which means $C[0] = C[1] = C[2] = C[3] = 0$ as desired.

### 7.2 Geometric Extrapolation

To show that the extrapolation notion is not limited to our [DMVY13] work we show the potential of its application in a case of geometric domains. Assume we need to determine whether one shape subsumes the other. It is fairly easy to find a solution if our setting is that of Fig. 7.6, where the shapes are squares with sides
parallel to the axis. We ask whether the left side of shape A is to the left of the left side of shape B, the top side of shape A is above the top side of shape B, etc.

\((A.lSide.x \leq B.lSide.x) \land (B.rSide.x \leq A.rSide.x) \land (B.tSide.y \leq A.tSide.y) \land (A.bSide.y \leq B.bSide.y)\). But what happens for more complex domains? What happens for instance, if we know that the domain consists of shapes whose sides can be described by \(f(a, b, c, d) = (x + y = a, x - y = b, x + y = c, x - y = d)\) \((a < c, b < d)\), on the \(x,y\) plane (as in Fig. 7.7)?

A simple solution, using our knowledge of the solution in the domain where the squares are parallel to the axes, would be to use a transformation. A linear transformation will do in this case, but any domain transformation that preserves the convexity of shapes will be suitable for the framework. This observation provides us with a rich set of domains in which we can check the subsumption of two shapes. Using the knowledge of how to determine square subsumption, we obtained in the domain where squares are parallel to the axes, we can extrapolate the proof.

A direct application of this suggestion can be found in verification using numerical domains. A box domain is relatively easy to calculate and reason about yet its precision is limited. For instance, it will loose precision when attempting to capture the possible values of our rotated figure example. Other domains such as Octagon and Polyhedra will not lose precision in this case, yet they are more costly to evaluate. The suggested approach may enable the use of the Box domain without loss of precision.

**Related Work** In [APS10] a domain of parallelotopes was defined and implemented. But comparison of speed was not possible due to the different settings used to implement Box and the new domain. We propose to use the box domain directly; doing so will reduce the verification effort substantially and allow use of the box domain as a black box.
Chapter 8

Conclusions and Possible Future Directions Rising from This Work

In the work described in this thesis, we explored various approaches to verification and, for cases where verification fails, various approaches to synthesis of correct synchronization placements.

In all our approaches, we assumed program correctness under sequential consistency and performed verification and synthesis by accounting for behaviors unique to the specific memory model. We note that, if the program was incorrect under sequential consistency, all of our approaches would discover it. This is a property of the memory models and synchronization mechanisms we explored: applying the strictest synchronization would limit the program to only sequentially consistent behaviors. For programs with only sequentially consistent behaviors, our tools can only be as strong as the state of the art for sequentially consistent verification, which in itself is a difficult problem. In this thesis, we did not discuss methods for strengthening sequentially consistent verification, but some of our insights can be useful there as well. One example would be the abstraction relaxations from Chapter 4 (as we explain in detail in Section 8.2).

We explored two kinds of Relaxed Memory Models: hardware (Intel’s x-86 TSO, and PSO) and software memory models (C++RMM). The latter proved to be more challenging. Since C++RMM is designed to be an abstraction of all possible hardware memory models that could be underneath it, it is more relaxed than any of the hardware memory models, and allows a wider set of behaviors for program execution.

As synchronization mechanisms, we synthesized either memory fences or memory order parameters to instructions. Under the C++ Relaxed Memory Model, memory order parameters allow more control over program behaviors than fences. For example, putting SC-fences between each two instructions in IRIW (from Section 6.4.2) will not prevent C++RMM relaxed behavior (see [Bat14, Section 2.8]).

We explored abstractions such as Predicate Abstraction and Numerical Abstract Domains, which allowed us to reason about infinite state programs. Applying abstraction to C++RMM is left for future research, since the domain proved to be challenging as is.
We were successful in verification, and synthesis of minimal synchronization, for challenging algorithms. These algorithms include: (i) a range of mutual exclusion algorithms (e.g. Dekker, Bakery, and Peterson), (ii) work stealing queues (Chase, THE, FIFO, etc.), (iii) a new version of RCU (d-prcu in Chapter 6), (iv) and other algorithms taken from literature. Some of the algorithms were infinite state, (e.g. Bakery), and others though finite stated proved to be challenging due to their intricate and interesting behaviors (e.g Dekker, Peterson, and WSQ). For each of our explored approaches, we designed tools that are publicly available.

We will next go over a few of the possible future directions that arise from our work.

8.1 Possible Directions Following our Predicate Abstraction Work

Chapter 3 used at its basis boolean program based predicate abstraction. Another approach to predicate abstraction is called Lazy Abstraction [HJMS02].

The process of Lazy Abstraction is as follows: After choosing an initial set of predicates, an abstract state space is explored until reaching an error (or verifying successfully). Then an abstraction refinement step is performed – Counter-Example Guided Abstraction Refinement (CEGAR). The path from the initial state of the abstract state graph, to the error state is simulated in the concrete domain. This simulation checks that the path is spurious, i.e., doesn’t exist in the concrete domain. A non-spurious path means the program has a real error, which is returned to the user. For a spurious path, a set of conditions is found that was missing from the abstraction and captures why that path is not feasible. This new set of conditions/predicates is added to the abstract state space along that path. The algorithm then continues to explore the changed graph for additional paths to an error state.

This process could be applied directly to a RMM setting after encoding the program to SC (the encoding, as we did in our [DMVY13] work). Two potential caveats for successful verification will be: (1) the costly CEGAR (simulation of abstract trace) step, which involves costly calls to a theorem prover; (2) the returned set of predicates might make the state space grow arbitrarily large.

Boosting the intermediate phases of Lazy Abstraction Instead of applying Lazy Abstraction as a black box, we can try to help it. After finding an abstract RMM trace to an error state during extrapolation, we can:

1. Replace CEGAR First check whether that abstract RMM trace is reachable for SC (in the abstract state space, reduced to SC feasible paths).
   - If the abstract trace is reachable under SC (spurious under SC proof), then we can extrapolate information from the SC CEGAR phase and prevent the more expensive RMM CEGAR step for that trace altogether.
Weakening using interpolants

Figure 8.1: Extrapolation with Interpolation

2. If not reachable then this trace is due to RMM effects. This will give us information about what predicates to add. Predicates refinement at this stage will be CEGAR between the SC abstract state space and the RMM abstract state space; refining that part of the abstraction that captured RMM effects with insufficient precision.

3. Assist CEGAR If previous attempts failed, or we don’t have an SC abstract trace, and we end up performing the CEGAR phase for RMM, we can try using only the SC predicates extrapolated from the original proof, and minimize the resulting state space.

Using extrapolation we can boost Lazy abstraction during its run.

Boosting the starting phase of Lazy Abstraction We can take the graph from SC Lazy Abstraction, extrapolate predicates for each node, and provide them as a starting point for Lazy Abstraction. We believe that adding extrapolation in this manner will prove to be both a speed-up to Lazy Abstraction for RMM and an enabling approach.

Proof Weakening using Interpolation

Given a pair of formulas \((\varphi^-, \varphi^+)\) such that \((\varphi^- \land \varphi^+)\) is unsatisfiable, an interpolant for \((\varphi^-, \varphi^+)\) is a formula \(\psi\) such that: 1. \(\varphi^-\) implies \(\psi\), 2. \(\psi \land \varphi^+\) is unsatisfiable, and 3. the variables of \(\psi\) are common to \(\varphi^+\) and \(\varphi^-\). The work of [HJMM04] showed that the use of interpolants allows an extraction of a minimal predicate abstraction proof. Our work on extrapolation of predicates brings a unique opportunity to both extend the scale of our work (since the number of predicates naturally affects the performance of our technique); more importantly, it allows us to explore the meaning of minimality of proofs. Our experience showed that the importance of predicates lies not only in their quantity but also in their quality. For RMMs, a small set of predicates can become a large set after extrapolation if the former addresses more shared variables. Moreover, a relatively large initial set of predicates might not grow as much during extrapolation.
Furthermore, for one of our examples, one of the initial predicate sets we worked with was not sufficient for the extrapolation procedure to successfully verify the program under RMM. This suggests that maybe that initial predicate set did not correctly capture properties important for verification.

We propose a framework described in Fig. 8.1. Given two semantics $\mathcal{L}_1 \subseteq \mathcal{L}_2$, and a proof under $\mathcal{L}_1$, we would like to extrapolate that proof (upper left dot) to a proof under semantics $\mathcal{L}_2$. But, instead of using extrapolation on the original proof, we propose to weaken it first using interpolants. We will do so by: taking a conjunct of: (i) negation of the property we need to prove, and (ii) the SC proof — the latter containing the predicates, invariants, and cubes (transitions from the boolean program) at each point. The long conjunct we receive is an un-satisfiable term from which we can derive interpolants for each point in the program. We can then extrapolate those interpolant formulas and use them as a basis for RMM proof of the program. We believe that this direction will lead to some interesting conclusions.

8.2 Possible Directions Following our Numerical Abstract Domains Analysis Work

In Chapter 4 we dealt with the challenge of infinite state algorithms. A common approach to addressing infinite state verification with numerical abstract domains is widening. One example would be a for-loop with a non-deterministic (even if known to be finite) number of iterations. During each pass of the loop the analysis state has to reflect the loop iterator value. Since that value grows at each iteration, a straightforward static analysis approach might not converge. To achieve convergence (or expedite it) widening is commonly used. At some point during the analysis, widening will abstract the currently observed abstract value of the iterator, changing it to be either the top possible value or to a value between its current and the top. Widening is necessary since the algorithms we want to verify are unbounded but it might lose intermediate invariants crucial to successful verification (further explained in Appendix 8.2.1). For instance, the program in Fig. 1.1 did not verify using ConcurInterproc (for any widening delay). If we bound both loops to 4, verification succeeds for widening delay 100 and fails for widening delay 2. The verification analysis can explore the difference in the invariants it receives, for the different widening delays, and encode them in the program, using assume instructions (which we try to avoid) or boolean variables.

8.2.1 Why Widening Hurts Verification?

Widening under the convex polyhedra domain is a complex issue and we will not elaborate on it here. For our purposes we will consider it as removing a side of an n-dimensional shape (n being the number of variables in the program). Given a program with variables $\{v_i\}_{1}^{n}$, a bound of the form $a_0 + \sum_{i=1}^{n} a_i v_i \geq 0$ for some $\{a_i\}_{0}^{n} \in \mathbb{Z}$ is
removed. In addition, to improve performance, some invariants in the state are stored implicitly. If we have $x - y \geq 0$ and $y \geq 0$, the invariant $x \geq 0$ will not be stored, but it is implied from the state. So widening may remove a complex invariant such as $y_1 - 8y_2 + 7T_0@index + 7T_1@index + 4 \geq 0$ because $index$ (the index of the bounded loop) changed from iteration to iteration; however if we look at what is implied from the entire state we will see that $y_1 \geq 0$ is lost.

8.2.2 How to Do Analysis Debugging Using Widening Delay

We think a beneficial approach will be to:

1. bound the program to a low number of iterations;
2. experiment with widening delay until a delay threshold is found,
   (i) above which the verification succeeds (widening doesn’t have the chance to remove invariants crucial to us)
   (ii) and below it fails (due to widening);
3. compare the states of successful and failed verification attempts and take the invariants that appeared in the successful attempt but not the failed one;
4. add those invariants with boolean variables.

Initial results seem to be promising. A simple extension of this approach will be to use interpolants, which are described in Section 8.1. Interpolants can minimize the invariant obtained during the process.

8.3 Possible Directions Following our C++ Relaxed Memory Model Synchronization Synthesis Work

The discussion in Chapter 6 was limited to finite state and finite execution length programs. These limitations rise from the underlying verification tool we used, CDSChecker[ND13, ND16], which at time of this writing remains state of the art in enumeration of program executions under the C++ Relaxed Memory Model. Several approaches exist to extending the state of the art.

8.3.1 Violation Patterns and Avoidance Templates are Not Complete

Make them complete with respect to corrections that use SC memory order parameters  A relation ws (or mo in other literature) orders all stores to a single memory location. In [Alg12], Alglave et al. define an fr relation which connects a load, $r$, reading from a write, $w_0$, to all the stores that succeed $w_0$ in the ws order (depicted in Fig. 8.2). They then proceed to define and prove that an execution that is acyclic
Figure 8.2: Visualization of the fr relation (ws orders all writes to a single memory location.)

over \( rf \cup fr \cup ws \cup po \) is sequentially consistent ([Alg12, Definition 21 and theorem 3]. Though the authors do not explicitly discuss the C++ Relaxed Memory Model, their results apply to it. We can use this results in Chapter 6. When we find a counter example that is acyclic when projected on \( rf \cup fr \cup ws \cup po \), that counterexample cannot be corrected with adding SC memory order parameters, and that error trace exists under sequentially consistent executions as well. We can use \( rf \cup fr \cup ws \cup po \) cycles to make our set of patterns complete with respect to SC.

**Make them complete with respect to corrections that use Rel-Acq memory order parameters** Similar reasoning can be applied with respect to the rel-acq axiomatic model. By using the rel-acq model definition from [BDG13] or [AMT14] by adding mo to our reasoning about hb (to the RD pattern) we may be able to guarantee that the absence of cycles (or paths over hb and one mo edge) means the error at hand is rel-acq equivalent. That is, it exists under executions where all instructions memory order parameters are rel-acq (or SC), and thus is not preventable with this form of synchronization.
Bibliography


[Pow] IBM Power ISA v.2.05. 2007.


96


ממציא הרעיון של "מערך זיכרון מרונים" 

מטרה של המאמר היא להציג את הטכניקה של "מערך זיכרון מרונים" והありがとうinya לำן תהליך הפעלת הסיסטה הנכונה. המאמר מציג תירות פנימה של תוצאות שמקבלות את המсадים של מחשבים מבית התוכנה של תוכנה המוחלת, ליבר תוכנות והתקנות ממקדים עלי הוספת סכין הרחיב גניט נוכ흰 לתוכנות.

תורמים וחברוגים:

1. אלגוריתמים פנימייםمبرית סכין של TSO/PSO
2. אלגוריתמים פנימייםمبرית סכין של TSO/PSO
3. סכין סכין מסייע לפקודת זיכרון עבבב

בנוסף מתנוהרגים עם מ“And-Or" פקודות פנימיות אט התורמים של.

Technion - Computer Science Department - Ph.D. Thesis PHD-2017-04 - 2017
The document is in Hebrew and is a continuation of a previous discussion on global variables and synchronization in computer science. The text is a research paper discussing the safety of a specific algorithm in a parallel processing environment. It explores the conditions under which the algorithm is safe to execute and the implications of different scenarios on the system's behavior. The text includes mathematical expressions and technical terms such as "safety" and "correctness". The discussion is complex and technical, requiring a strong background in computer science to fully understand.
תקציר


Figure 8.3: A simplified version of Lamport’s Bakery. (Initiall Y1=Y2=0)

אמוני מדיניות מוסמן של אנגלרים הממאפיין של ליפמקיוס, שיווי מוסון האﻹגלריים בחר

אמוני מדיניות מקיוע ענודת זכרון

בא lặngדוסמ תופי של התהליך, כל זה מהם תכנית לקטוךكرויים של (5) תכנית התהליך

שת嘡ליים שני מוחים לקטוך קרויים. על מנה שניה אצ ומדירוגים בשני מוחים שלטים (1)
The research was conducted under the guidance of Professor Eran Yahav, in the Faculty of Computer Science. Part of the research results were published in articles written by the author of the dissertation, as well as in conference proceedings and journals throughout his Ph.D. research period, which are updated versions:


todot

The research is based on the work conducted with the guidance of Eran Yahav, and the author expresses his appreciation to the following:

Anir, to his parents, Yelena and Michael, who supported him throughout the process. Anir also acknowledges the help of his close colleagues in the faculty, for the entire process was conducted with their support. And thanks to all those who contributed to the dissertation.

The author expresses his gratitude to Technion for the support of his research.

The research was conducted under the guidance of Eran Yahav, in the Faculty of Computer Science. Part of the research results were published in articles written by the author of the dissertation, as well as in conference proceedings and journals throughout his Ph.D. research period, which are updated versions:


The author expresses his gratitude to Technion for the support of his research.
אקסטרפולציה וсинטזה تحت מודלים זרוכיים

ascar הקיר

ليبורי על מחקר

לשם مليי חלקי של הדרישות לעבלת התואר

דוקטור בפילוסופיה

ורי שמשון

הונג לסלן הוכנונן – מכון טכנולוגי לישראל

חיים הרצינג

נובמבר 2016
אקסטרפולציה וсинטזה تحت מודל זכרון

嘞להים

יוריר מוסמן