Preserving Correctness
Under Relaxed Memory Models

Michael Kuperstein
Preserving Correctness Under Relaxed Memory Models

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Michael Kuperstein

Submitted to the Senate of the Technion — Israel Institute of Technology
Shvat 5772 Haifa February 2012
This thesis was completed under the supervision of Dr. Eran Yahav (Department of Computer Science, Technion) and Dr. Martin Vechev (IBM T.J. Watson Research Center and ETH Zurich) in the Department of Computer Science.

First and foremost, I’d like to thank my advisors, Eran and Martin. Eran, Martin, I feel I was extremely lucky to have worked with you. I could not have hoped for a better pair of advisors. Or a better pair of new, true, friends.

I would like to thank Greta Yorsh, Noam Rinetzky and Mooly Sagiv for their insightful and valuable comments on early drafts of the papers that eventually became this thesis. I would also like to thank Nir Shavit for suggesting to Eran the basic idea from which this thesis evolved.

If I were to thank all of my friends at the Technion personally, each for their own support and friendship, it would take ups the better part of this thesis. I would, however, like to dedicate a special thank-you to my officemates, Yohai Kaplan and Gadi Aleksandrowicz. You made the Technion a fun place to come to every day.

Last and not least, I’d like to thank my parents — for everything.

The generous financial support of the Technion and IBM Research is gratefully acknowledged.
# Contents

Abstract

1 Introduction

2 Preliminaries and Notation

3 Relaxed Memory Models
   3.1 Introduction
   3.2 The RLX memory model framework

4 Fence Inference
   4.1 Constraint Generation
   4.2 Instantiation for RLX
   4.3 Limitations

5 Abstract Memory Models
   5.1 Introduction to Abstract Interpretation
   5.2 Partial Store Order
   5.3 Fence Inference under PSO
   5.4 Partial Coherence Abstractions
   5.5 Abstract Fence Inference

6 Experimental Results
   6.1 Fence Inference in the RLX framework
   6.2 Abstract Memory Models

7 Related Work

8 Future Work
# List of Figures

1.1 Pseudo-code of the Chase-Lev work stealing queue .......................... 4

3.1 Categorization of relaxed memory models (from [3]) .................. 14

3.2 Operational semantics defining transition from $(G, L, E, B, op)$
to $(G', L', E', B', op')$ ................................................. 18

4.1 An example program (a) and its partial transition system (b).
Avoidable transitions are drawn with thicker lines ..................... 30

5.1 Example program with unbounded state-space under relaxed
models ............................................................................. 47

5.2 Peterson’s Algorithm with explicit memory operations ........... 52

5.3 Store buffers for Peterson’s algorithm of Fig. 5.2 under the
PSO memory model ............................................................. 52

5.4 Operational semantics defining a transition from $(G, L, pc, B)$
to $(G', L', pc', B')$ under PSO .......................................... 53

5.5 LOAD-G rule for concrete TSO ........................................... 55

5.6 Abstraction of a single buffer, parameterized by $k$ ............... 59

5.7 Abstract semantics defining a transition from $(G, L, pc, B)$ to
$(G', L', pc', B')$ .............................................................. 64

5.8 Diagram showing relationships in Lemma 5.4.2 ....................... 66

5.9 Peterson’s Algorithm with fences that guarantee mutual ex-
clusion under the PSO memory model .................................. 70

5.10 May-contain flush semantics ............................................. 74

6.1 Enqueue operation of the Michael-Scott queue (from [15]) ..... 84

6.2 A version of Lamport’s fast mutex algorithm for 2 processors.
The code given is for process $i$ .......................................... 91
Abstract

This thesis addresses the problem of automatic verification and fence inference in concurrent programs running under relaxed memory models. Modern architectures implement relaxed memory models in which memory operations may be reordered and executed non-atomically. Instructions called memory fences are provided to the programmer, allowing control of this behavior. To ensure correctness of many algorithms, the programmer is often required to explicitly insert memory fences into her program. However, she must use as few fences as possible, or the benefits of the relaxed architecture may be lost. It is our goal to help automate the fence insertion process.

We present a framework for automatic inference of memory fences in concurrent programs, relieving the programmer from this complex task. The framework consists of two parts:

- An algorithm that given a finite-state program, a safety specification and a description of the memory model computes a set of ordering constraints that guarantee the correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution violating the specification. These constraints are then realized as additional fences in the input program.

- A family of novel partial-coherence abstractions, specialized for relaxed memory models. These abstractions allow us to extend the applicability of the algorithm to programs that are infinite-state under the relaxed memory model, even when they were finite-state under the “standard” sequentially consistent model.

We implemented our approach in a pair of tools called FENDER and BLENDER and used them to infer correct and efficient placements of fences for several non-trivial algorithms, including practical mutual exclusion primitives and concurrent data structures.
Chapter 1

Introduction

In 1979, in his seminal paper “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs” [43], Leslie Lamport defined the “sequential consistency” (SC) criterion for correctness of multiprocessor computers. Such a computer is called sequentially consistent if:

*The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual processor appear in the sequence in the order specified by its program.*

It was Lamport’s intent that any correct multiprocessor computer implementation must meet this criterion. However, modern hardware architectures are not, in fact, sequentially consistent. Instead, they implement so-called “relaxed” (or “weak”) memory models (RMMs) [3]. These models enable improved hardware performance compared to sequentially consistent hardware [29]. This is achieved by allowing the CPU and memory subsystems to perform memory operations out of order and non-atomically. Unfortunately, this behavior poses an additional burden on the programmer. Even when the underlying architecture is sequentially consistent, highly-concurrent algorithms are notoriously hard to get right [56]. When programming for an architecture that implements an RMM, programmers must also reason about executions that have no sequential analogue. This reasoning is non-intuitive and may lead to subtle concurrency bugs.

To allow programmers avoid non-sequentially consistent executions, architectures provide special *memory fence* (also known as *memory barrier*)
instructions. Very informally, a fence instruction restricts the CPU and memory subsystem’s ability to reorder operations, thus eliminating some undesired non-SC executions. Finding a correct and efficient placement of memory fences for a given concurrent program is a challenging task. Using too many fences (over-fencing) hinders performance, while using too few fences (under-fencing) may allow unexpected incorrect executions to occur. Manually balancing between over- and under-fencing is very difficult, time-consuming and error-prone (cf. [33, 14, 15]). Furthermore, the process of finding fences has to be repeated whenever the algorithm changes, and whenever it is ported to a different architecture.

As an example, consider the problem of implementing the Chase-Lev work-stealing queue [20] (“CL”) on a relaxed memory model. Work stealing is a popular mechanism for efficient load-balancing used in runtime libraries for languages such as Java, Cilk [7] and X10 [19]. Fig. 1.1 shows an implementation of this data structure in C-like pseudo-code. For now, ignore the fence instructions that appear on unnumbered lines. CL maintains an expandable array of items called \texttt{wsq} and two indices \texttt{top} and \texttt{bottom} that can wrap around the array. The queue has a single owner thread that can only invoke the operations \texttt{push()} and \texttt{take()} which operate on one end of the queue, while other threads may call \texttt{steal()} to take items out from the opposite end. For simplicity, we assume that items in the array are integers and that memory is collected by a garbage collector (manual memory management presents orthogonal challenges, see [54]).

We would like to guarantee that there are no out of bounds array accesses, no items are lost (by being overwritten before being read), and no “phantom” items are read after being removed. All these properties hold for the CL queue under the sequentially consistent memory model. However, they may be violated when it is used under a relaxed model.

Under the SPARC RMO [36] memory model (which we cover in more detail in Chapter 3), some of the memory operations in the code may be executed out of order. Tab. 1.1 shows possible RMO re-orderings that lead to violation of the specification. The column \texttt{locations} lists the two lines in a given method which contain memory operations that might get reordered and lead to a violation. The next column gives an example of an undesired effect when the operations at the two labels are reordered. The last column shows the type of fence that can be used to prevent the undesirable
```c
typedef struct {
    long size;
    int *ap;
} item_t;

long top, bottom;
item_t *wsq;

void push(int task) {
    long b = bottom;
    long t = top;
    item_t *q = wsq;
    if (b - t ≥ q->size - 1) {
        q = expand();
    }
    q->ap[b % q->size] = task;
    fence("store-stores");
    bottom = b + 1;
}

int take() {
    long b = bottom - 1;
    item_t *q = wsq;
    fence("load-load");
    long t = top;
    if (b < t) {
        bottom = t;
        return EMPTY;
    }
    task = q->ap[b % q->size];
    if (b > t) {
        return task;
    }
    if (!CAS(&top, t, t + 1))
        return ABORT;
    bottom = t + 1;
    return task;
}

int steal() {
    long t = top;
    fence("load-load");
    long b = bottom;
    fence("load-load");
    item_t *q = wsq;
    if (t ≥ b) {
        return EMPTY;
    }
    task = q->ap[t % q->size];
    fence("load-store");
    if (!CAS(&top, t, t + 1))
        return ABORT;
    return task;
}

item_t* expand() {
    int newsize = wsq->size * 2;
    item_t* newitems = (item_t*) malloc(newsize*sizeof(int));
    item_t* newq =
        (item_t*) malloc(sizeof(item_t));
    for (long i = top; i < bottom; i++) {
        newitems[i % newsize] = wsq->ap[i % wsq->size];
    }
    newq->size = newsize;
    newq->ap = newitems;
    fence("store-store");
    wsq = newq;
    return newq;
}
```

Figure 1.1: Pseudo-code of the Chase-Lev work stealing queue
reordering. Informally, the type describes what kinds of memory operations have to complete before other type of operations. For example, a store-load fence executed by a processor forces all stores issued by that processor to complete before any new loads by the same processor start.

For a more detailed example of the effect the memory model has on execution, we consider the failure described in line 2 of Tab. 1.1. This corresponds to a reordering of operations at lines 4 and 5 in the `take()` method: if these two lines are reordered, the read from `top` is executed before the write to `bottom`. The failure scenario involves one process running the `steal()` method in parallel to another processes running a sequence of `take();push();take();push()` as follows:

1. Initially the queue has one item with `top = 0` and `bottom = 1`.
2. A `take()` reads `top` and gets preempted before executing line 6.
3. An entire `steal()` executes and correctly returns the item at index 0 and advances `top` to 1.
4. The `take()` resumes and succeeds, returning the same item as the previous `steal()`, setting `bottom` to 0.
5. A complete `push()` now pushes some item `i`.
6. A complete `take()` executes and returns `EMPTY` instead of item `i`.
7. A complete `push()` executes and overwrites item `i` (losing item `i`).

To guarantee correctness under RMO, the programmer can try to manually insert fences that avoid undesirable behavior. As an alternative to placing fences based purely on her intuition, the programmer may also use a tool such as CheckFence [15] that can check the correctness of a given fence placement. However, repeatedly adding fences to avoid each counterexample can easily lead to over-fencing: a fence used to fix a counterexample may be made redundant by another fence inferred for a later example. In practice,
localizing a failure to a single reordering is challenging and time consuming
as a single failure trace might include multiple instances of non-SC behavior.
Furthermore, a single reordering can be exhibited as multiple failures, and it
is sometimes hard to identify the cause underlying an observed failure trace.

In a nutshell, the programmer is required to manually produce Tab. 1.1:
summarize and understand all counterexamples from a checking tool, local-
ize the cause of failure to a single reordering, and propose a fix that elimi-
nates the counterexample. Further, this process might have to be repeated
manually every time the algorithm is modified or ported to a new mem-
ory model. Even a subtle change in the algorithm may require a complete
re-examination.

It is easy to see that the process of manual fence inference does not scale.
In this thesis we present an algorithm that automatically infers correct and
efficient fence placements for finite-state programs. Our inference algorithm
is defined in a way that makes the dependencies on the underlying memory
model explicit. This makes it possible to use our algorithm with various
memory models. To demonstrate the applicability of our approach, we im-
plemented a relaxed memory model that supports key features of several
modern RMMs.

Requiring the input program to be finite-state means we must overcome
several challenges for the algorithm to be practical. First, this requirement
means the algorithm, taken as is, is not suitable for fence inference in open
systems (such as library implementations). This is in contrast to our goal to
apply the algorithm to concurrent data structures. To formally verify that a
data structure meets a specification (and, consequentially, to infer a correct
fence placement), one generally needs to verify the “most general client” [4]
which is usually not finite-state. We deal with this in a manner similar to
other related work in the field (e.g Burckhardt et al. [15]) by using represen-
tative clients. Another problem is that even if a program is finite state
under sequential consistency it will often not be finite-state under a relaxed
model. As this phenomenon is common in practice, a direct implementation
of our algorithm fails to infer fences for many interesting programs. To solve
this problem we developed the concept of abstract memory models (AMMs).
Very informally, an abstract memory model is an over-approximation of a
relaxed memory model, in the sense that any program behavior possible
in the RMM is also possible in the abstract model. Our abstract memory
models are designed so as a program that is finite-space under SC remains finite state under the AMM. By utilizing AMMs, we can use our algorithm for any program that is finite-space under SC.

To recap, our main contributions are:

- A family of abstract memory models that enable finite-state modeling of concurrent programs for a wide class of memory models.

- A novel algorithm that automatically infers a correct and efficient placement of memory fences in finite-state concurrent programs.

- A prototype implementation of the abstractions and algorithm in a tool capable of inferring fences under several memory models.

- An experimental evaluation of the prototype it can infer fences for several state of the art concurrent algorithms, including popular lock-free data structures and synchronization primitives.

The rest of this thesis is organized as follows: Chapter 2 provides some preliminaries and notation used in the thesis. Chapter 3 gives an overview of relaxed memory models, and presents the RLX memory model framework. The main focus of our work appears in Chapters 4 and 5. Chapter 4 describes our memory fence inference algorithm and instantiates it in the context of a specific memory model framework. Chapter 5 describes abstract memory models and an adaptation of our inference algorithm to those models. Chapter 6 shows the experimental results of our tools. Chapter 7 surveys previous work done on the subject of verification and fence inference under RMMs. Finally, Chapter 8 describes possible directions for future work.

The thesis is based on work previously published as [40] and [41].
Chapter 2

Preliminaries and Notation

As the mother teaches her children how to express themselves in their language, so one Gypsy musician teaches the other. They have never shown any need for notation. — Franz Liszt

In this chapter we introduce the basic concepts and notation used throughout the thesis. A reader familiar with the topics discussed may safely skip the chapter, and continue on to Chapter 3.

Sequence Notation

Given a finite set \( D \), we use \( Seq_n(D) \) to denote the set of all sequences of length \( n \) over \( D \), \( Seq_{\leq n}(D) \) to denote the set of all sequences shorter than or equal in length to \( n \) over \( D \), \( Seq(D) \) to denote the set of all finite sequences over \( D \), \( |w| \) to denote the length of a sequence \( w \) and \( \epsilon \) to denote an empty sequence. We denote the concatenation of two sequences \( w_1, w_2 \) by \( w_1 : w_2 \). For \( 0 < k \leq |w| \), we define \( head(w,k) \) as the subsequence consisting of the first \( k \) items in \( w \) and \( tail(w,k) \) as the subsequence consisting of the last \( k \) items in \( w \). For \( k > |w| \) we define \( head(w,k) = tail(w,k) = w \) and for \( k \leq 0 \), \( head(w,k) = tail(w,k) = \epsilon \). We define \( last(w) \) to be the only element in \( tail(w,1) \). If \( tail(w,1) = \epsilon \) then \( last(w) \) is defined to be the special designated element \( \perp \) (which is not part of \( D \)). We define \( Set(w) \) to be the set of elements in the sequence \( w \). Finally, we define \( UTail(w,k) \) as \( Set(tail(w,|w| - k)) \) — the set of all but the first \( k \) elements of \( w \).
Order Theory

In this thesis, we use some standard definitions from order theory, which are repeated here for convenience. Let $D$ be a partially ordered set with the partial order $\sqsubseteq$. Let $X \subseteq D$ be some finite subset of $D$. We say $u \in D$ is an upper bound of $X$ if $\forall x \in X. x \sqsubseteq u$. Let $U(X)$ be the set of upper bounds of $X$. $l$ is the least upper bound (LUB) of $X$ if $\forall u \in U(X). l \sqsubseteq u$. The definitions of a lower bound and greatest lower bound (GLB) are symmetrical. We denote the LUB of $X$ by $\bigsqcup X$ and the GLB by $\bigsqcap X$. We define the join and meet operators over $D$, $x \sqcup y$ and $x \sqcap y$ as $\bigsqcup \{x, y\}$ and $\bigsqcap \{x, y\}$ respectively. We say $D$ is a join-semilattice if for any $x, y \in D$, the least upper bound $\bigsqcup \{x, y\}$ exists. Note that the existence of all pair-wise LUBs implies the existence of an LUB for any finite subset of $D$. Again, a meet-semilattice is defined symmetrically. We say $D$ is a lattice if it is both a join- and a meet-semilattice. Finally, we say $D$ is a complete lattice if there exists an LUB and a GLB for any (not necessarily finite) subset $X \subseteq D$. Note that by definition every finite lattice is also complete. We also define $\bot = \bigsqcap D$ and $\top = \bigsqcup D$ (the bottom and top) elements, respectively. These elements are guaranteed to exist if $D$ is a complete lattice (or a finite semi-lattice of the appropriate type).

One final definition we require from order theory is that of a Galois connection. Let $L$ and $M$ be lattices, and $\alpha : L \to M$, $\gamma : M \to L$ functions between those lattices. We say $\alpha$ and $\gamma$ form a Galois connection if for all $l \in L, m \in M$, we have:

$$\alpha(l) \sqsubseteq_M m \iff l \sqsubseteq_L \gamma(m)$$

Boolean Functions and Propositional Formulae

There are several standard ways to define boolean functions. We use the notation described below. We use $E = \{tt, ff\}$ to denote the standard boolean domain, with the standard order relation ($ff \sqsubseteq tt$) and conjunction/disjunction operators. Let $D$ be an arbitrary finite set. A boolean function over $D$ is a function $f : \mathcal{P}(D) \to E$. Equivalently, we could define a boolean
function as \( f : E^D \rightarrow E \), if we identify the set \( X \subseteq D \) with the function:

\[
i_X = \lambda d \in D. \begin{cases} 
\text{tt} & \text{if } d \in X \\
\text{ff} & \text{otherwise}
\end{cases}
\]

A boolean function is **monotone** if for any \( X, Y \subseteq D \), \( X \subseteq Y \implies f(X) \subseteq f(Y) \). For a given set \( D \) the set of all monotone boolean functions is known to form complete lattice. The order relation on this lattice is defined as: \( f \sqsubseteq g \) if \( \forall X \subseteq D. f(X) \subseteq g(X) \). The meet and join are defined member-wise, that is \( (f \sqcap g)(X) = f(X) \land g(X) \), \( (f \sqcup g)(X) = f(X) \lor g(X) \). Note that the lattice above isomorphic to the free distributive lattice over \( D \) \[6\]. It is a basic fact that every Boolean function over \( D \) can be expressed as a propositional formula over \( D \), and there exists some representation \( \psi_f \) of the function in conjunctive normal form (CNF). Additionally, if \( f \) is a monotone function, then \( \psi_f \) is a **monotone propositional formula**: all variables in appear in \( \psi_f \) in positive form. Due to this equivalence between boolean functions and propositional formulae, throughout this thesis we use functions and propositional formulae interchangeably. When convenient, for boolean functions \( f, g \) we write \( f \land g \), \( f \lor g \) and \( f \implies g \) to stand for \( f \sqcap g \), \( f \sqcup g \) and \( f \subseteq g \) respectively. Similarly, we write \( \delta \models f \) instead of the more precise \( f(\delta) = \text{tt} \) or \( \delta \models \psi_f \).

**Program Syntax and Semantics**

We define a *program* \( P \) in the standard fashion, as a tuple consisting of:

- An initial state, \( \sigma_0 \).
- A set of process identifiers, \( \text{PID} = \{0, \ldots, n\} \).
- The program code \( \text{Prog}_i \) for each process \( i \).
- Two disjoint sets of variables: a set of local variables \( \text{Locals} \) and a set of shared variables \( \text{Shared} \).

The program code is a sequence of (uniquely) labeled statements, where the labels are taken from some set \( \text{Labs} \). The uniqueness requirement may be relaxed, where appropriate.

The statements our programming language uses are shown in Tab. 2.1. The table also shows the intuitive meaning of each statement. The precise semantics depend on the chosen memory model, and are described in chapters 4 and 5. Note that in the examples given in the thesis we usually do
Table 2.1: Syntax of assembly-like programming language

not use this low-level language, but rather present the examples in higher level (usually C-like) pseudo-code. The translation from the pseudo-code into the low-level language is straight-forward.

We treat the state of a program is an opaque object — as its precise definition depends on the definition of the programming language semantics. For now, it is useful to think of the program state as a valuation of the variables and program counters for each of the processes in PID.

A transition system (TS) is a tuple \( \langle \sigma_0, \Sigma_P, T_P \rangle \), where \( \Sigma_P \) is a set of states, \( \sigma_0 \in \Sigma_P \) is a designated initial state, and \( T_P \) is a set of transitions \( \sigma \rightarrow \sigma' \) where \( \sigma \in \Sigma_P \) is the source state, and \( \sigma' \in \Sigma_P \) the destination state. We denote the source state of a transition \( t \) by \( \text{src}(t) \) and the destination state by \( \text{dest}(t) \). Informally, a transition \( t \) from \( \sigma_1 \) to \( \sigma_2 \) appears in the transition system if \( \sigma_1 \) appears in it, and the program semantics allow a transition between the two states. If a transition \( t \) corresponds to the execution of some program statement with label \( l \), we apply this label to transition, denoted as \( t = \sigma \xrightarrow{l} \sigma' \). Transitions that do not correspond to program statement may be labeled differently or unlabeled, depending on the semantics. If a transition \( t \) is associated with a process (e.g. it is a statement executed by that process), we denote this associated process ID as \( \text{proc}(t) \). A trace of the program \( P \) is a path in \( \langle \sigma_0, \Sigma_P, T_P \rangle \), that is a (possibly infinite) sequence of transitions \( t_0, t_1, ..., t_i, ... \) s.t. the source state of \( t_0 \) is \( \sigma_0 \) and the source state of \( t_{i+1} \) is the target state of \( t_i \).

Formally, to be able to construct transition systems, we need to encapsulate program semantics in a transformer function. Let \( C \) be the set of all possible program states (as opposed to actually reachable states). The transformer \( \tau : \mathcal{P}(C) \rightarrow \mathcal{P}(C) \) is defined according to the program semantics. For a single state \( \sigma \), \( \tau(\{\sigma\}) \) is defined to be set of all legal (according
to the semantics) successor states of $\sigma$. In this case, we will sometimes omit the braces and write $\tau(\sigma)$. For a non-singleton set $\Sigma \subseteq C$, we define $\tau(\Sigma) = \bigcup \{\tau(\sigma) \mid \sigma \in \Sigma\}$. We can then define the transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$ for a program $P$ as:

$$\Sigma^k_P = \tau^k(\sigma_0)$$

$$\Sigma_P = \bigcup_{n \in \mathbb{N}} \Sigma^n_P$$

$$T_P = \{\sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \in \Sigma_P \land \sigma_2 \in \tau(\sigma_1)\}$$

**Verification and Synthesis**

In extremely general terms, the field of software verification strives to solve the following problem: given a program $P$ and a specification $\psi$, check whether $P$ satisfies $\psi$ ($P \models \psi$). Usually, the specification $\psi$ is provided in the form of a temporal logic [64] formula. The related field of program synthesis deals with the problem of creating $P$ such that $P \models \psi$ for a given specification $\psi$. Finally, program repair deals with modifying a given program $P \not\models \psi$ into a new program $P'$ such that $P' \models \psi$.

The two most common types of specifications of interest are those that require safety or liveness properties to hold. Informally, a safety property means that “nothing bad ever happens” while a liveness property means that “something bad eventually happens”. In this thesis, we only deal with the verification (and repair with respect to) safety properties. This restriction allows us to take the following simple view on program correctness.

Let $\psi$ be a propositional formula over the program state. That is, for a given state $\sigma$, we can directly evaluate whether $\sigma \models \psi$. If $\sigma \not\models \psi$, we say $\sigma$ is an error state. We can now say that the program satisfies the specification ($P \models \psi$) iff $\Sigma_P$ does not contain any error states. Note that this is equivalent to $P \models G\psi$ when the specification is described in LTL, but as we deal only with safety properties, we can omit the $G$ without any risk of confusion.
Chapter 3

Relaxed Memory Models

Consistency is contrary to nature, contrary to life. The only completely consistent people are the dead. — Aldous Huxley

3.1 Introduction

The sequentially consistent model, as defined by Lamport[43], allows programmers to assume their programs have “interleaving semantics”. That is, programs appear to behave as if memory operations execute atomically and are interleaved, with the instructions of each process appearing in the trace in their original program order. This is no longer true for relaxed memory models. According to the categorization by Adve et al. [3] (shown in Fig. 3.1), there are five key relaxations with respect to SC that relaxed memory models implement - and all of them break the illusion of interleaving. These relaxations can be roughly grouped into “order relaxations” and “store atomicity relaxations”. The three leftmost columns in Fig. 3.1 represent order relaxations:

- $W \rightarrow R$: A memory model that implements this relaxation may delay a memory store until a later (in program order) load from a different location is performed.
- $W \rightarrow W$: This relaxation allows a memory store to be delayed until a later (in program order) store to a different location is performed.
- $R \rightarrow RW$: This relaxation allows a memory load to be delayed until any later memory operation to a different location is performed.
None of these relaxations allow reordering of memory operations that access the same memory location. Thus memory coherence is not violated.

The two columns on the right represent store atomicity relaxations — that is, whether a store can be seen by a process before it is globally performed. “Read own writes early” means a process may observe stores it has itself performed before they are visible to other processes. “Read others’ writes early” means that a store by process $p$ may be observable to only a subset of the other processes, while not being observable to all of them. However, as before, the relaxations cannot violate memory coherence. Examples of code affected by each of these relaxations can be found in [3].

It is important to keep in mind that Fig. 3.1 is not a fully accurate description of the features of the various memory models. For example, the Alpha memory model is quite different from Sparc RMO, even though they support the same basic relaxations, as Alpha sometime allows data-dependent load instructions to be (observably) reordered. One notable model missing from Fig. 3.1 is the Intel x86 (IA32) model. A good overview of the x86 model can be found in the work of Sarkar et al. [69]. More recent work by the same group (Owens et al. [61]) shows x86 is equivalent to Sparc TSO.

The two major types of formal definitions for memory models are axiomatic and operational definitions. The axiomatic style describes the memory model in terms of axioms which define the set of legal memory traces of a program. Using axiomatic definitions, it is relatively easy to check whether a given trace is legal under the memory model. However, these definitions do not provide an easy way to generate the possible memory traces of a program from an initial memory and program state, and thus are not suitable for explicit-state model checking.

---

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>Order Relaxations</th>
<th>Atomicity Relaxations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$W \rightarrow R$</td>
<td>$W \rightarrow W$</td>
</tr>
<tr>
<td></td>
<td>$R \rightarrow RW$</td>
<td>$R$ Others’ W Early</td>
</tr>
<tr>
<td></td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>SC</td>
<td>IBM 370</td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>PSO</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Alpha</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>RMO</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>PowerPC</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
</tbody>
</table>

Figure 3.1: Categorization of relaxed memory models (from [3])
The operational style describes the memory model in terms of operational semantics of the system. This has the opposite effect - it is suitable for model checking, but not for verifying whether a given memory trace is legal. This type of definitions can be further divided into two subtypes:

- In reordering-based semantics, instructions are executed out of program order. These semantics usually use a reordering table (e.g. the “bypass table” of Yang et al. [81] or “instruction reordering table” of Shen et al. [71]) to control which pairs of instructions may be reordered.

- In buffer-based semantics, instructions are executed in their original order. However, memory reads and writes have non-standard semantics. Those semantics are described in terms of buffer-like structures. Examples of such definitions can be seen in work by Mador-Haim et al. [49] and, in a more practical form, in [61].

The semantics of memory fences must also change depending on the memory model formalization. Generally, for reordering-based semantics, a fence controls the order of execution. That is, if a fence is placed between instructions inst1 and inst2, then inst1 must be executed before inst2. In buffer-based models, a fence (very informally) forces memory operations to take global effect. A more detailed explanation of fence semantics for buffer-based models appears in Chapter 5.

The three most commonly provided types of fences are “store-load”, “store-store” and “load-load” fences. The semantic meaning of those types of fences is best understood in the context of reordering-based models: for example, a store-load fence prevents an earlier store from being reordered with a later load. More generally, given an instruction sequence P1, followed by a store-load fence, which is in turn followed by instruction sequence P2, all stores in P1 must take (global) effect before any loads in P2 may execute.

Hardware platforms also provide “combination” fences, that enforce more than one type of order. For example, a full memory barrier is a fence that combines the effect of all 3 types above.

This thesis deals only with hardware-level memory models, that is, deviations from SC caused by the hardware. There is also a large body of research on the subject of language-level models (used by high-level programming languages such as Java). Those models present separate challenges, and are beyond the scope of our current work, mostly due to their inherent com-
plexity. An introduction to the challenges of language-level models can be found in the work of Boehm [9].

3.2 The RLX memory model framework

In this section, we describe a framework for operational reordering based semantics. The framework is based on previous work by Park & Dill [62], Shen et al. [71] and Yang et al. [81], and allows us to formally handle the TSO and PSO memory models. We also use it to formalize a model slightly stronger than RMO — the differences between our version and standard RMO will be discussed later.

The syntax of a simple assembly-like programming language we use for our programs was shown in Tab. 2.1. This syntax was chosen so that every instruction can make at most one memory access. In many real assembly languages, a single instruction may cause several accesses, which means instructions must be broken down into several memory events (e.g. like in [69]). Restricting instructions to a single memory access significantly simplifies the semantics. More importantly, it allows a fence to be placed between any two memory accesses. On the other hand, we allow arbitrary arithmetic and boolean expressions over local variables (registers) to be used in the branch and localop statements. The semantics of expressions over locals do not change between memory models. To simplify presentation we do not give semantics for heap memory allocation, or access to this type of memory (e.g. indirection). In our implementation we use a standard naive deterministic implementation of the heap [65], and each heap location is treated as a global variable. We explore heap-related issues in some more detail in Chapter 6.

In the RLX framework, a state of a program $\sigma$ is defined as a tuple $\langle G_\sigma, L_\sigma, E_\sigma, B_\sigma, op_\sigma \rangle$ where:

- $G_\sigma : \text{Shared} \rightarrow \mathbb{Z}$ is a valuation of the shared variables.
- $L_\sigma : \text{PID} \rightarrow (\text{Locals} \rightarrow \mathbb{Z})$ is a valuation of the local variables of each process.
- $E_\sigma : \text{PID} \rightarrow \text{Seq}(\text{Labs})$ maps a process to a sequence of labels that represents the operations waiting to be executed by the process.
- $B_\sigma : \text{PID} \rightarrow (\text{Shared} \rightarrow \text{Seq}(\mathbb{Z}))$ maps a process and a shared variable to a sequence of values.
• $op_\sigma \in Labs \cup \{\bot\}$ is the label of the instruction selected to be executed next.

We omit $\sigma$ when it is clear from context.

Fig. 3.2 shows the semantics of the programming language, in operational style. In all rules but SELECT, $p$ is used as a shorthand for $\text{proc}(op)$. An inference rule is enabled if a state $\sigma$ satisfies the premise of the rule. If the rule is enabled, then the state $\sigma'$ which satisfies the conclusion of the rule is a successor state of $\sigma$, that is $\sigma' \in \tau(\sigma)$.

The syntax of the conclusions in Fig. 3.2 (and our other semantics figures) has slightly nonstandard notation. For example, the conclusion of the LOCAL OP rule should be read as $L_{\sigma'} = L_{\sigma}(p)[r \mapsto e]$. Additionally, in all rules in the figure, with the exception of the SELECT rule, it is silently assumed that the conclusion also contains $op' = \bot$.

In our semantics, $E_\sigma(p)$ represents the execution buffer of process $p$. This is a sequence of instructions that are potentially eligible for execution by process $p$ in state $\sigma$. This is equivalent to the “reordering box” used in [62] and “local instruction buffer” in [81]. The semantic rules that affect $E(p)$ are BRANCH-T and BRANCH-F. The BRANCH rule adds a linear sequence of instructions (denoted by $lseq()$) to the buffer. If a branch to label $tgt$ is taken, then the added sequence is the sequence of instructions from $tgt$ up to the next branch instruction. If the branch is not taken, then the added sequence starts at the instruction immediately following the branch in the program code (denoted by $\text{next}(op)$). In any case, the instructions are added to the buffer in their original program order. The SELECT rule selects and removes an instruction for execution from the buffer, thereby determining the next enabled rule for a given processor. This means that given a processor $p$ and a sequence $t_1, t_2, t_3, ...$ of transitions associated with $p$, all odd-numbered transitions are SELECT and even-numbered transitions represent actual execution. The SELECT rule exists purely for convenience of presentation: selection and execution can be merged, without ill effect, into a single rule, but this would clutter presentation of the semantics. On the other hand, the SELECT transitions become cumbersome when discussing transition systems. From this point on, when considering transition systems generated by these semantics, we merge the selection and the execution of an operation into a single transition, and omit the $op$ state component completely.
\[
\begin{align*}
op &= \bot & l \in \text{enabled}(\sigma) & p = \text{proc}(l) \quad \text{(SELECT)} \\
op' &= l & E'(p) = E(p) \setminus l \\
\text{stmt}(op) &= \text{fence} & \text{nop} \\ 
\text{stmt}(op) &= \text{load } r \leftarrow x & B(p)(x) = \epsilon & v = G(x) \\ & & L'(p)(r) = v \\ 
\text{stmt}(op) &= \text{store } x \leftarrow r & B(p)(x) = b & v = L(p)(r) \\ & & L'(p)(r) = v \\ 
\text{stmt}(op) &= \text{flush } x & B(p)(x) = v \cdot s \\ & & L'(p)(r) = v \\ 
\text{stmt}(op) &= \text{if } c \text{ branch } tgt & L(p)(c) = \mathbf{tt} & \text{BRANCH-T} \\ & & E'(p) = E(p) \cdot \text{lseq}(\text{tgt}) \\ 
\text{stmt}(op) &= \text{if } c \text{ branch } tgt & L(p)(c) = \mathbf{ff} & \text{BRANCH-F} \\ & & E'(p) = E(p) \cdot \text{lseq}(\text{next}(op)) \\ 
\text{stmt}(op) &= q \leftarrow \text{cas } x, r, s & G(x) = L(p)(r) & L(p)(s) = v \\ & & G'(x) = v & L'(p)(q) = \text{true} \\ 
\text{stmt}(op) &= q \leftarrow \text{cas } x, r, s & G(x) \neq L(p)(r) & \text{CAS-F} \\ & & L'(p)(q) = \text{false} \\ 
\text{stmt}(op) &= \text{localop } r \leftarrow e & L(p)(e) = v & \text{LOCALOP} \\ & & L'(p)(r) = v 
\end{align*}
\]

Figure 3.2: Operational semantics defining transition from \(\langle G, L, E, B, \op \rangle\) to \(\langle G', L', E', B', \op' \rangle\)
While the semantics of RLX are for the most part reordering-based, to support non-atomic stores, we also require a store-buffer component. When a store $x \leftarrow v$ instruction is added to the execution buffer, it is split (similarly to the CRF model, Arvind et al. [71]) into a sequence of two instructions: $\text{store } x \leftarrow v; \text{flush } x$. The store instruction adds the value $v$ into the buffer $B(p)(x)$, and flush moves it into local memory. Buffering values in this fashion lets local load operations read values that are not yet visible to other processors if they are ordered after the corresponding store but before the flush. This split allows us to talk about the model purely in terms of reordering, without paying any additional attention to store atomicity. We will elaborate on store buffering in Chapter 5.

As shown in the premise of the SELECT rule, an operation can be selected for execution only if it is enabled. Intuitively, an operation is enabled if it is in the execution buffer, and there is nothing preventing it from being executed. There are two reasons an operation in the execution buffer may be ineligible for execution: 1. The memory model does not allow it to be executed out of order with respect to operations preceding it in the buffer. 2. There is a data dependency between it and a preceding operation.

Formally, let $I_{\sigma}(p)$ be $\text{Set}(E_{\sigma}(p))$ — the set of instruction in the execution buffer of process $p$. We say an operation $op$ at label $l_1$ is enabled in state $\sigma$, denoted by $l_1 \in \text{enabled}(\sigma)$, if all of the following conditions hold:

- $l_1 \in I_{\sigma}(p)$
- $\forall l_2 \in I_{\sigma}(p). l_2 <_{\sigma,p} l_1 \implies \text{bypass}(\text{stmt}(l_2), \text{stmt}(l_1))$
- $\forall l_2 \in I_{\sigma}(p). l_2 <_{\sigma,p} l_1 \implies \neg \text{conflict}(\text{stmt}(l_2), \text{stmt}(l_1))$

In this definition, we introduce three instances of new notation: $<_{\sigma,p}$, conflict and bypass, which we will now explain.

The partial order $<_{\sigma,p}$ (the dynamic program order) is defined as follows. $l_1 <_{\sigma,p} l_2$ if:

$$\exists i \forall j. (E_{\sigma}(p)(i) = l_1 \land E_{\sigma}(p)(j) = l_2) \implies i < j$$

Less formally, $l_1 <_{\sigma,p} l_2$ if $l_1$ is in the execution buffer of $p$, and the first occurrence of $l_1$ is before any occurrence of $l_2$ in the buffer. This definition may seem strange at first sight, but has a clear rationale. In many formal
weak memory models, the existence of a “program order” is assumed. The program order is clearly well-defined for loop-free programs - it is the order in which statements appear in the program. However, when programs that contain loops are considered, program order must be defined more carefully (cf. Shen et al. [71]). For example, an instruction at label $l_1$ may be after $l_2$ in one iteration of the loop, but before $l_1$ is executed again in the next iteration. Dynamic program order captures the expected program order from any point in the execution: $l_1 <_{\sigma,p} l_2$ precisely if in-order execution by process $p$ starting from state $\sigma$ would first execute $l_1$ and then $l_2$.

\textit{conflict}(st_1, st_2) holds if there is a data dependency that prevents $st_2$ from being re-ordered with $st_1$. More concretely, $st_1$ and $st_2$ are in conflict if they access the same variable (either shared or local) and at least one of them writes a value to that variable. Note that a \textit{load} instruction is considered a write to a local variable. \textit{flush} instructions are never considered in conflict with any other instruction. In the extended version of our semantics that supports the heap and indirect accesses, we may not always know in advance which memory location will be accessed by an instruction. In this case we conservatively consider it in conflict when required.

\textit{bypass}(st_1, st_2) determines whether statement $st_2$ is allowed to “bypass” $st_1$ according to the memory model semantics. By $st_2$ bypassing $st_1$ we mean that $st_1$ precedes $st_2$ in the dynamic program order, but $st_2$ is eligible for execution. For example, if $st_1$ is a fence statement, then no other statement may bypass it, so $\textit{bypass}(st_1, st_2)$ will always be false. The bypass tables we have used to represent TSO, PSO and RMO are Tab. 3.1, Tab. 3.2 and Tab. 3.3 respectively. A “Yes” mark means that if the row instruction $st_1$ is followed by the column instruction $st_2$, $\textit{bypass}(st_1, st_2)$ evaluates to true, and a “No” mark means it evaluates to false. A “DiffVar” mark means the bypass is enabled only for different memory locations. This is used to preserve memory coherence: a \textit{FLUSH} may never bypass an earlier \textit{FLUSH} for the same variable, but under some models flushes for different variables may be performed out of order. The “N/A” mark in the \textit{BRANCH} row means the bypass predicate is never evaluated with a branch as the first instruction. This is an artifact of RLX semantics: if a branch instruction appears in the buffer, it is always the last instruction, since instructions are only added past a branch when the branch is removed.

Note that this model of RMO is in fact slightly stronger (less relaxed)
<table>
<thead>
<tr>
<th>LOAD</th>
<th>STORE</th>
<th>CAS</th>
<th>L.OP</th>
<th>BR</th>
<th>FENCE</th>
<th>FLUSH</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>STORE</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CAS</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>L.OP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>FENCE</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FLUSH</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOAD</th>
<th>STORE</th>
<th>CAS</th>
<th>L.OP</th>
<th>BR</th>
<th>FENCE</th>
<th>FLUSH</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>STORE</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CAS</td>
<td>No</td>
<td>Yes</td>
<td>DiffVar</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>L.OP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>FENCE</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FLUSH</td>
<td>Yes</td>
<td>Yes</td>
<td>DiffVar</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOAD</th>
<th>STORE</th>
<th>CAS</th>
<th>L.OP</th>
<th>BR</th>
<th>FENCE</th>
<th>FLUSH</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>STORE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CAS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>L.OP</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>FENCE</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FLUSH</td>
<td>Yes</td>
<td>Yes</td>
<td>DiffVar</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
than the official RMO specification (given in the SPARC Architecture Manual [36]). There are two features of RMO that we do not support. The first is speculative loads. An RMO system is allowed to perform control speculation, and execute loads on the predicted control path. If the branch turns out to have been predicted correctly the loaded value can be used. In our model those executions are impossible, since branches are only resolved once all of their dependencies are satisfied.

The second issue is that we consider all data hazards to be conflicts, even if there is no real dependency. This, however, is not strictly correct. For example, consider the so-called write-after-write hazard. This is a situation in which two subsequent loads load different memory locations to the same register. In our model, this is considered a conflict, and the two loads cannot be reordered. However, on a real system, the hazard could be avoided (e.g. by using register renaming), and reordering could occur. A combination of this issue with the previous one occurs when dealing with conflicts that involve unresolved memory locations. While our model treats this case conservatively, official RMO allows performing loads under the speculative assumption that no conflict exists.

Since it is not our goal to precisely model RMO, and we only use the RLX framework as an example memory model on which we can implement our algorithm, we can accept this inaccurate representation.
Chapter 4

Fence Inference

The more modern type of reformer goes gaily up to a fence and says, “I don’t see the use of this; let us clear it away.” To which the more intelligent type of reformer will do well to answer: “If you don’t see the use of it, I certainly won’t let you clear it away. Go away and think. Then, when you can come back and tell me that you do see the use of it, I may allow you to destroy it.” — G.K. Chesterton

4.1 Constraint Generation

In this section, we present our inference algorithm in a general setting, without instantiating it for a specific memory model. We then prove that when properly instantiated, it is correct and optimal.

Goal

Before we present our algorithm, we must state more concretely what the goal of the algorithm is. The input to the algorithm is a finite-state program $P$, a safety specification $S$, and an operational description of the memory model $M$. We assume that $P$ satisfies $S$ under sequential consistency but not necessarily under $M$. The output of the algorithm is a new program $P'$, that satisfies $S$ under $M$, which is obtained by adding memory fences
to \( P \). For “reasonable”\(^1\) memory models, this problem will always have a trivial solution, as placing a fence between every two memory operations will reduce the possible executions to those allowed under SC. Therefore, we also add an optimality constraint: we would like the program \( P' \) to have performance as close to the original program as possible. In other words, since there may be many possible ways to “fix” \( P \) by adding fences, we want to choose only the best solution(s).

**Algorithm Structure**

The problem given above can be broken down into the following sub-problems:

1. Construct the transition system \( \langle \sigma_0, \Sigma_P, T_P \rangle \) for \( P \) under \( M \).
2. Find the set of “error states” \( E \) violating \( S \) in \( \langle \sigma_0, \Sigma_P, T_P \rangle \).
3. Compute a set of program locations in \( P \) s.t. adding fences in those locations would “cut off” the error states, and output a program \( P' \) with fences added in these locations.

The high-level structure of our algorithm follows these three steps. Steps 1 and 2 are standard in the world of software verification (in particular, software model checking [21]). The focus of this work, and the “heart” of our algorithm is therefore step 3. The general approach we use (similarly to the work of Vechev et al. [79]) is not to try to compute fence locations directly, but use an intermediate constraint language.

The high-level idea is that we first choose a constraint language \( F \) and associate with every transition \( t \) some constraint \( \chi(t) \) from \( F \). Very informally, we say that a constraint \( \chi(t) \) is *enforceable* if we can “cut the transition \( t \) off” from the transition system by adding (syntactic) fences to the program. We call actually adding such fences *enforcing* the constraint.

We can use this idea to break the problem down into the 3 following major sub-steps: constraint *generation*, *solving* and *implementation*.

1. (Generation) Compute a boolean formula \( \psi \) over the constraints that represents all of the ways to cut off all error states in \( E \) (that is, make them unreachable in \( \langle \sigma_0, \Sigma_P, T_P \rangle \)).
2. (Solving) Find a minimal satisfying assignment to \( \psi \). This gives us a minimal constraint set \( \delta \) that, if enforced, will cut off all error states.

\(^1\)A more rigorous treatment of “reasonable” or “SC-Counter” memory models is given in Chapter 5
Note that it is possible that there are several such minimal constraint sets.

3. (Implementation) Transform \( \delta \) into a fence placement that enforces those constraints. Again, there may be many ways to implement the constraints as fences.

For this scheme to work, we need to compute \( \psi \) such that every satisfying assignment (constraint set) \( \delta \) of \( \psi \) satisfies the following three properties:

- Every constraint in the set \( \delta \) are enforceable using memory fences inserted into the program code of \( P \).
- If all constraints in \( \delta \) are enforced by inserting memory fences into \( P \) (creating a new program \( P' \)), then \( P' \) does not violate the specification \( S \).

Additionally, we want the computed constraint formula \( \psi \) to be maximally permissive: a constraint set satisfies the formula if and only if enforcing it will make the modified program adhere to the specification. This means a minimal satisfying assignment \( \delta \) of \( \psi \) represents a “globally minimal” constraint set: it is impossible to fix the program by enforcing only a strict subset of \( \delta \).

**Transition System Construction**

The first stage of the algorithm is to construct the transition system (TS) for the program. As explained in Chapter 2 the transition system is a graph that consists of vertices which represent the states that can be reached by running the program and edges that represent state transitions. The transitions link a state to all of its possible successor states. In this chapter, we assume the input program \( P \) is *finite-state under the memory model*. This means that when \( P \) is executed under the memory model, there is only a finite number of reachable states. Note that if a program is finite-state under sequential consistency, this does not imply it is finite-state under a more relaxed model. We will discuss this distinction in detail in Chapter 5.

Since our algorithm is designed to work with *operational* memory models, given a program state \( \sigma \) we are able to directly compute its set of successors. This means we can construct the transition system iteratively, using a standard worklist algorithm [39].
Marking Error States

Once the TS is constructed, we can identify a subset of error states: the set of states that violate the provided safety specification. It is well known [52] that every LTL safety property \( \phi \) can be expressed as \( Gp \) where \( p \) is a “past-formula”, that is, a formula that only refers to the past of the computation. In this thesis, we further assume that that the specification is given as \( Gp \) where \( p \) is a state property - for example, an assertion on the values of program variables. This restriction is introduced for two reasons:

1. While our algorithm is sound in the general case, it is no longer necessarily optimal. This is because for a general past-formula, fixing the program may be possible not by cutting off the error states themselves, but by cutting off some of their predecessors. It is possible that our algorithm can be extended to this case, but we did not explore this possibility in details.

2. This restriction allows us to check whether a state is an error state immediately after we encounter it during exploration. This allows us not to explore any of the error states’ descendants, which improves the algorithm’s performance.

As many practically useful safety properties can be expressed as \( Gp \) where \( p \) is a state property, we believe this to be a reasonable restriction.

Constraints

Our main goal is to transform an input program \( P \) into a new output program \( P' \) that satisfies \( S \). However, at this stage, it is convenient to “abstract away” the two programs and focus purely on transition systems. Given a transition system \( \langle \sigma_0, \Sigma_P, T_P \rangle \) under a memory model \( M \), we can identify some transitions as avoidable and others as unavoidable. A transition \( t = \sigma_1 \rightarrow \sigma_2 \) is considered avoidable if it is possible to construct a program \( P' \) by adding fences to \( P \) s.t. \( \langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle \) does not contain a transition that corresponds\(^2\) to \( t \). Since discussing two separate transition systems (for \( P \) and \( P' \)) is cumbersome, we informally refer to this process as cutting \( t \) from the transition system \( \langle \sigma_0, \Sigma_P, T_P \rangle \).

More practically, we need to pick (according to the memory model) a set of constraints such that every such constraint can be enforced by adding

\(^2\)The precise formal meaning of “corresponds” in this context will be given later.
memory fences to \( P \). We then associate with every transition \( \sigma_1 \xrightarrow{t} \sigma_2 \) a set of constraints \( \chi(t) \), that satisfies the following properties:

- If at least one of the constraints in \( \chi(t) \) is enforced, then \( \sigma_2 \) is no longer reachable from \( \sigma_1 \) in \( P' \).
- If none of the constraints are enforced, and \( \sigma_1 \) is reachable in \( P' \), then \( \sigma_2 \) is also reachable.

In other words, \( \chi(t) \) precisely captures all the ways to cut the transition \( t \).

This means a transition is avoidable if and only if \( \chi(t) \) is non-empty. We can “lift” this definition from transitions to program traces and states: An avoidable trace is a trace in which at least one transition is avoidable, and avoidable state is a state such that all program traces leading to it are avoidable.

Given a transition system and a specification we wish to find a constraint set which would cut all traces leading to error states. One possible approach, in the spirit of previous work by Vechev et al. [79], is to enumerate all (acyclic) traces leading to an error state and try to prevent each trace by enforcing appropriate constraints. However, such enumeration does not scale to practical programs as the number of traces can be exponential in the size of the transition system which is itself exponential in program length. Instead, our algorithm works on a state-by-state basis by assigning an avoid formula to each state. The avoid formula of a state captures all the ways to cut that state from the transition system.

Suppose we want to cut the state \( \sigma \). Let the incoming transitions of \( \sigma \) be \( t_1, \ldots, t_k \), with source states \( \sigma_1, \ldots, \sigma_k \) respectively. To cut \( \sigma \), we must make it unreachable through all of its incoming transitions. For each transition \( t_i \), this means either cutting \( t_i \) itself or removing the source state \( \sigma_i \). More concretely, we must either enforce some constraint in \( \chi(t_i) \) or recursively find the avoid formula for \( \sigma_i \) and enforce some satisfying assignment of that formula. This is in fact recursive only if the transitions system \( \langle \sigma_0, \Sigma_P, T_P \rangle \) is acyclic - if it contains cycles, the avoid formula of \( \sigma_i \) may itself depend on the avoid formula of \( \sigma \). This suggests that the desired avoid formula for a state is a fixed point of a function that relates the avoid formula of a state to those of its predecessor states.
**Constraint Generation Algorithm**

Now that the definitions are in place, we can present the algorithm used for the constraint formula generation phase. Instead of dealing directly with formulae, we will for convenience present the algorithm in terms of boolean functions ("avoid functions"). We will, however, often abuse notation and identify boolean variables with atomic propositions, and monotone boolean functions with monotone propositional formulae that define those functions. In particular:

- For a function $f$ and an assignment of values to variables $\delta$ we will use $\delta \models f$ to mean that $f(\delta) = \text{tt}$.
- For functions $f_1, f_2$, we use $f_1 \implies f_2$ to mean $f_1 \sqsubseteq f_2$, $f_1 \lor f_2$ to mean $f_1 \sqcup f_2$, etc.

Let $V$ be a set of variables, representing possible constraints on execution. Let $F$ be the set of monotone boolean functions over $V$ with the standard order relation (also known as the free distributive lattice over $V$). Let $\langle \sigma_0, \Sigma_P, T_P \rangle$ be a transition system and $\sigma_0$ the initial state. Then a legal labeling function is a function $L : \Sigma_P \rightarrow F$, such that $L(\sigma_0) = \text{ff}$. Intuitively, the labeling function $L$ attaches an avoid function to a state. We require $L(\sigma_0)$ to always be false as the initial state can never be avoided. For a given transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$, we denote by $\Lambda_P$ the set of all legal labeling functions for that transition system.

Given a labeling function $L$ and a state $\sigma \in \Sigma_P$, we define:

$$\text{avoid}(L, \sigma) = \bigwedge \{ L(\mu) \lor \chi(t) \mid (\mu \xrightarrow{t} \sigma) \in T_P \}$$

This formalizes the previously presented intuition: given a labeling $L$, to avoid a state $\sigma$ we must avoid all incoming transitions $\mu \xrightarrow{t} \sigma$, either by cutting $t$ (using $\chi(t)$) or by enforcing $L(\mu)$. In the above definition, we abuse notation by treating $\chi(t)$ not as a subset of $V$, but rather as the function represented by $\bigvee \{ p \mid p \in \chi(t) \}$. We will continue this abuse throughout this and the following chapter. Whether $\chi$ is treated as a set or a function is always clear from the context.

Using $\text{avoid}$ we define an operator $\text{trans} : (\Sigma_P \rightarrow F) \rightarrow (\Sigma_P \rightarrow F)$ that updates the labeling to the “next generation” of avoid functions:

$$\text{trans}[L] = \lambda \sigma \in \Sigma_P. L(\sigma) \land \text{avoid}(L, \sigma)$$
If $L$ is legal, then so is $\text{trans}(L)$ because:

- $\text{trans}[L](\sigma_0) = (L(\sigma_0) \land ...) = \text{ff}$
- $\text{avoid}(L, \sigma)$ is monotone, as is $L(\sigma)$, and a conjunction of two monotone functions is also monotone.

The algorithm to find the desired labeling function is now very simple: we take the initial labeling function $L_0$ defined below, and iteratively apply $\text{trans}$ until a fixed point is reached.

$$L_0 = \lambda \sigma \in \Sigma_p. \begin{cases} \text{ff} & \text{if } \sigma = \sigma_0 \\
\text{tt} & \text{if } \sigma \neq \sigma_0 \end{cases}$$

From this point on, we refer to the $L$ function to which the fixed point computation converges as $av$.

However, directly applying this algorithm is inefficient, for two reasons. First, it requires maintaining two copies of the transition system. More importantly, a lot of unnecessary computation is performed because it is possible that in every application of $\text{trans}$ only few $L(\sigma)$ values actually change. Therefore we use an optimized version based on the standard “chaotic iteration” method due to Cousot & Cousot [23]. This version is shown in Algorithm 1.

Lines 2-4 of the algorithm set the initial labeling to $L_0$. The labeling is then updated in the following fashion. First, the entire transition system is added to a workset. Then, if the worklist is not empty we pick an arbitrary state $\sigma$, and update it from $L(\sigma)$ to $\text{trans}[L](\sigma)$ (lines 8 - 10). We then check whether $L(\sigma)$ was changed by the application of $\text{trans}$. If it has, we may need to update the labeling of its descendant states, so we add all descendants of $\sigma$ to the workset. When the workset becomes empty, a fixed point has been reached, so we can return the conjunction of constraints for the error states.

**Example**

Consider the simple concurrent program shown in Fig. 4.1(a). $X$ and $Y$ are integer variables shared between processes A and B, while $R_1$ and $R_2$ are integer variables local to process B. For illustrative purposes, the memory model we use here is a simplified version of RLX where stores to global memory are atomic and the allowed relaxation is reordering data indepen-
R1 = R2 = X = Y = 0;

A:
A1: STORE X = 1
A2: STORE Y = 1

B:
B1: LOAD R1 = Y
B2: LOAD R2 = X

(a)

Figure 4.1: An example program (a) and its partial transition system (b). Avoidable transitions are drawn with thicker lines.
Algorithm 1: Constraint Generation

Input: Program P, Specification S, Memory Model M
Output: Program P' satisfying S

1. compute \( \langle \sigma_0, \Sigma_P, T_P \rangle \) under the memory model M
2. \( L(\sigma_0) \leftarrow \text{false} \)
3. \textbf{foreach} state \( \sigma \in \Sigma_P \setminus \{ \sigma_0 \} \) \textbf{do}
   
4. \( L(\sigma) \leftarrow \text{true} \)
5. \textbf{workset} \leftarrow \Sigma_P \setminus \{ \sigma_0 \}
6. \textbf{while} workset is not empty \textbf{do}
7. \( \sigma \leftarrow \text{select and remove state from workset} \)
8. \( \varphi \leftarrow L(\sigma) \)
9. \textbf{foreach transition} \( t = (\mu \rightarrow \sigma) \in T_P \) \textbf{do}
10. \( \varphi \leftarrow \varphi \land (L(\mu) \lor \chi(t)) \)
11. \textbf{if} \( L(\sigma) \not\equiv \varphi \) \textbf{then}
12. \( L(\sigma) \leftarrow \varphi \)
13. \( \text{add all successors of } \sigma \text{ in } \Sigma_P \text{ to workset} \)
14. \( \psi \leftarrow \bigwedge \{ L(\sigma) \mid \sigma \not\equiv S \} \)
15. \textbf{return} \( \psi \).

The constraint language we use in the example consists of constraints on execution order. The constraint \( [L_1 \prec L_2] \) where \( L_1 \) and \( L_2 \) are program labels means we forbid \( L_2 \) to bypass by \( L_1 \). Informally, \( [L_1 \prec L_2] \) means that if \( L_1 \) precedes \( L_2 \) in program order, then \( L_1 \) must be executed before \( L_2 \). We give a detailed explanation of the constraints used for the (full) RLX model later in this chapter.

Fig. 4.1(b) shows part of the transition system of this program running on this specific memory model. We only show states that can lead to an error state, as the rest of the transition system is not relevant to the example.

Inside each state in the figure we show: (i) assignments to the local variables of each process (\( L_1 \) and \( L_2 \)), and the global variables \( G \); (ii) the execution buffer of each process (\( E_1 \) and \( E_2 \)); (iii) the (final) avoid formula of the state. Since stores are atomic, we do not show \( B_\sigma \).

For this program, our specification is that \( R_1 \geq R_2 \) in the program’s final state. In the initial state (state 1) all four variables have the value 0. The transition system also contains a single error state (state 9) where \( R_1 = 0 \) and \( R_2 = 1 \) (state 9). Since the transition system is acyclic, we can
find $av(\sigma)$ by topologically sorting the states, and then computing $av$ once for each state. For example:

- Since state 1 is the initial state, the avoid formula is necessarily $\mathbf{ff}$.
- The avoid formula for state 2 is computed by taking the disjunction of avoiding the transition $A_2$ and avoiding the source state of the transition (state 1). To do so we first need to know $\chi(1 \rightarrow 2)$. Informally, we need to know whether $A_2$ is executed out of order, and which alternative instructions could have been executed by $A$ instead. If we examine the execution buffer $E_1$ of state 1 and look at the instructions that precede $A_2$, we find that $A_2$ is executed out of order, and that $A_1$ precedes it in the buffer. This implies we can enforce the constraint $[A_1 \prec A_2]$ as a way to avoid the transition $A_2$. Since the source state (state 1) cannot be avoided, the avoid formula for state 2 is simply $[A_1 \prec A_2]$. The formula $[B_1 \prec B_2]$ for state 3 is obtained similarly.
- The transition from state 2 to state 4 is taken “in order”, that is, it doesn’t violate any enforceable constraint. Therefore, the transition itself cannot be avoided and the only way to avoid reaching 4 is through enforcing the avoid formula of its predecessor, state 2. So the avoid formula of state 4 is also $[A_1 \prec A_2]$.
- State 5 has two incoming transitions: $B_2$ and $A_2$. $B_2$ is taken out of order from state 2 and can be prevented by enforcing the constraint $[B_1 \prec B_2]$. The constraint for the source state 2 is $[A_1 \prec A_2]$, so the overall constraint is $[B_1 \prec B_2] \lor [A_1 \prec A_2]$. Similarly, we perform the computation for transition $A_2$ from state 3 which generates an identical constraint. The final avoid formula for state 5 is thus the conjunction of $[B_1 \prec B_2] \lor [A_1 \prec A_2]$ with itself. In other words, it is $[B_1 \prec B_2] \lor [A_1 \prec A_2]$.
- For the error state 9, the two incoming transitions are executed in-order and cannot be avoided. The overall constraint is thus generated as a conjunction of the constraints of the predecessor states 7 and 8, and it is $[B_1 \prec B_2] \land [A_1 \prec A_2]$.

Note that since there is only one error state, the resulting overall formula is the avoid formula of that error state: $[B_1 \prec B_2] \land [A_1 \prec A_2]$. 

32
Handling Boolean Functions

The algorithm, as presented above, “hides” several representation and performance issues related to boolean functions. The clearest issue is that the algorithm returns a boolean function that represents a constraint formula. However, to actually place fences we require not the formula but rather its minimal satisfying assignments. We “offload” this task to standard SAT solving tools. As our experience (cf. Chapter 6) shows, the SAT-solving stage is not a performance bottleneck.

A bigger issue is the fact every step of the algorithm requires an equivalence check of two boolean functions (the test $L(\sigma) \neq \varphi$ in Line 11). This is NP-hard in general, and remains NP-hard even under the restriction that both functions are monotonic. With an explicit formulae representation those checks become very computationally expensive. However, if the functions are represented as Binary Decision Diagrams (BDDs) [12], then the equivalence check is, in our experience, not a practical bottleneck.

Specifications

As mentioned earlier, throughout this thesis we only deal with specifications of the form $Gp$ where $p$ is a state property. In practice, we wish to restrict the specification further. First, we do not allow specifications that depend on buffer contents. Rather, the specification language is formed only by predicates that depend on values of variables (local or global) and the program counter. This is a very natural restriction — we have no interest in specifications that directly refer to properties of, e.g., the execution buffer. A slightly less natural restriction is that we do not allow specifying values for global variables except in final states. This is due to the fact that in non-final states each process may have a different view of the global variables. Consider for instance the extremely simple program $A_1$: \texttt{STORE x = 1; A_2: LOAD v = x; A_3: END}, with $X$ being initially 0. After the statement $A_1$ executed, but before the stored value is flushed, $G(x) = 0$. However, if the \texttt{LOAD} at $A_2$ is executed at this point, $v$ will get the value 0. Hence it is unclear whether that intermediate state violates the assertion $x \neq 1$. We avoid this issue by disallowing this type of specifications. In the example above, the specification could be replaced with $pc = A_3 \land x \neq 1$ or with $v \neq 1$. While this limits the expressiveness of the specification language, we feel this limi-
tation to be in-line with our intuition of what specifications of code running under a relaxed model should look like.

Algorithm Correctness

To show our algorithm is correct, we need to demonstrate that (a) the fixed point computation terminates, and (b) once it terminates, enforcing the avoid formula $av(\sigma)$ indeed cuts the state $\sigma$. We first need several technical lemmas:

**Definition 4.1.1** If $L_1, L_2 \in \Lambda_P$, we say that $L_1 \sqsubseteq L_2$ if $\forall \sigma \in \Sigma_P. L_1(\sigma) \implies L_2(\sigma)$. We define $L_1 \sqcup L_2$ as $\lambda \sigma. L_1(\sigma) \lor L_2(\sigma)$ and $L_1 \sqcap L_2$ as $\lambda \sigma. L_1(\sigma) \land L_2(\sigma)$.

**Lemma 4.1.1** $\Lambda_P$ ordered by $\sqsubseteq$, with the join and meet operators $\langle \sqcup, \sqcap \rangle$ defined above is a finite complete lattice.

*Proof:* Holds by construction, as $\Lambda_P$ is essentially a product of $|\Sigma_P| - 1$ instances of $\mathbb{F}$ with the product order.

**Lemma 4.1.2** $\text{trans}$ is a continuous function over $\Lambda_P$.

*Proof:* Since $\Lambda_P$ is a finite lattice, it is enough to prove $\text{trans}$ is monotonic. Let $L_1, L_2 \in \Lambda_P$ s.t. $L_1 \sqsubseteq L_2$. We wish to show that $\text{trans}[L_1] \sqsubseteq \text{trans}[L_2]$. Let $\sigma \in \Sigma_P$ be some state. We then need to prove that $\text{trans}[L_1](\sigma) \sqsubseteq \text{trans}[L_2](\sigma)$. If $\sigma = \sigma_0$ then $\text{trans}[L_1](\sigma) = \text{trans}[L_2](\sigma) = \text{ff}$ and we are done. Otherwise, $\sigma$ has at least one predecessor $\mu$. Since $L_1 \sqsubseteq L_2$, For any $(\mu \xrightarrow{t} \sigma) \in T_P$, $L_1(\mu) \sqsubseteq L_2(\mu)$. This implies that $(L_1(\mu) \lor \chi(t)) \sqsubseteq (L_2(\mu) \lor \chi(t))$. Since this condition holds for any predecessor $\mu$, we have $\text{avoid}(L_1, \sigma) \sqsubseteq \text{avoid}(L_2, \sigma)$. And, finally, since $L_1(\sigma) \sqsubseteq L_2(\sigma)$ we have:

$$\text{trans}[L_1](\sigma) = L_1(\sigma) \land \text{avoid}(L_1, \sigma) \sqsubseteq L_2(\sigma) \land \text{avoid}(L_2, \sigma) = \text{trans}[L_2](\sigma)$$

**Corollary 4.1.3** Let $L^0 = L_0$ and $L^{k+1} = \text{trans}[L^k]$. Then there is some $k$ s.t. $L^{k+1} = L^k$, and this is the greatest fixed point of $L$.

---

3The number of elements in the product is not $|\Sigma_P|$ because $L(\sigma_0)$ is a constant.
Proof: Note that $L_0$ is the top element of $\Lambda_P$. This corollary then follows from Lemma 4.1.2, the fact $\Lambda_P$ is finite and the Knaster-Tarski fixpoint theorem [73].

\[ \square \]

**Corollary 4.1.4** The chaotic iteration version of the algorithm converges to the same result as directly applying $trans$.

Proof: Follows immediately from Lemma 4.1.2 and a standard result due to Cousot & Cousot [24].

This proves the version of the algorithm given in Algorithm 1 indeed converges to a fixed point. Since chaotic iteration necessarily converges to the same result as directly applying $trans$, the rest of this chapter will assume $trans$ is applied directly over the entire domain of $L$.

To formally prove the algorithm is correct, we first need to define the required correctness property slightly more precisely. Let $\delta \subseteq V$ be a set of propositions and $\phi \in F$. $\chi(t)$ is, by definition, the set of all constraints s.t. satisfying any of them will cut the transition $t$. If we interpret this set as the formula $\bigvee \{p \mid p \in \chi(t)\}$, this means that enforcing a constraint set $\delta$ cuts $t$ if and only if $\delta \models \chi(t)$. Similarly, for a trace $\pi = t_1, t_2, ..., t_k$, we define $\chi(\pi) = \bigvee \{\chi(t) \mid t \in \pi\}$ and say that enforcing a constraint set $\delta$ prevents $\pi$ if and only if $\delta \models \chi(\pi)$. Finally, $\delta$ cuts a state $\sigma$ if and only if for every trace $\pi$ that ends in $\sigma$, $\delta \models \chi(\pi)$. We use the notation $\delta \not\sim \sigma$ to mean that $\delta$ cuts $\sigma$, and $\delta \sim \sigma$ to mean it does not.

To see this definition of $\delta \not\sim \sigma$ is justified, assume that given an assignment $\delta$, we have some way of modifying the program $P$ into a new program $P_\delta$. This program has the transition system $\langle \sigma_0, \Sigma_P, T_\delta \rangle$ where $T_\delta = \{t \in T_P \mid \delta \not\models \chi(t)\}$. That is, all transitions that can be cut by enforcing $\delta$ have in fact been removed. We can now prove a very simple lemma:

\[ \text{Lemma 4.1.5} \]

A state $\sigma \in \Sigma_P$ is reachable in $\langle \sigma_0, \Sigma_P, T_\delta \rangle$ if and only if $\delta \not\sim \sigma$.

Proof: If $\delta \not\sim \sigma$ then there exists a trace $\pi$ to $\sigma$ (in $T_P$) s.t. $\delta \not\models \chi(\pi)$. This means that for all transitions $t$ in $\pi$, $\delta \not\models \chi(t)$. So all transitions in $\pi$ are in $T_\delta$, and $\pi$ is a trace in $\langle \sigma_0, \Sigma_P, T_\delta \rangle$. Conversely, if $\sigma$ is reachable in $\langle \sigma_0, \Sigma_P, T_\delta \rangle$, it contains a trace $\pi$ from $\sigma_0$ to $\sigma$. Since all of its transitions are in $T_\delta$, $\delta \not\models \chi(\pi)$, so $\delta \not\sim \sigma$. \[ \square \]
Having justified the definition, we can state the soundness property of Algorithm 1.

**Theorem 4.1.6** Let $\psi = \bigwedge \{ \text{av}(\sigma) \mid \sigma \not\in S \}$, and $\delta \subseteq \mathbb{V}$. If $\delta \models \psi$, then $\forall \sigma \in \Sigma_P. (\sigma \not\in S \implies \delta \not\models \sigma)$.

Informally, this theorem states that if we enforce any satisfying assignment of $\psi$ by removing the appropriate set of edges, then the resulting transition system will have no error states.

**Proof:** We first prove a lemma, from which the theorem follows:

**Lemma 4.1.7** Let $L$ be a fixed point of $\text{trans}$. For any finite trace $\pi$ from $\sigma_0$ to $\sigma$, if $\delta \models L(\sigma)$, then $\delta \models \chi(\pi)$.

**Proof:** We prove by induction on the number of transitions in $\pi$. For the case $k = 0$ (an empty trace), because $L(\sigma_0) = \text{ff}$, the statement is vacuously true.

For the case $k = 1$, the trace contains just one transition $\sigma_0 \xrightarrow{t_0} \sigma$. Since $L$ is a fixed point, $L(\sigma) \equiv L(\sigma) \land \text{avoid}(L, \sigma)$. So $\delta \models \text{avoid}(L, \sigma)$. $\text{avoid}(L, \sigma)$ is a conjunction, so we have $\delta \models L(\sigma_0) \lor \chi(t_0)$. Since $L$ is a legal labeling, $L(\sigma_0) = \text{ff}$, so $\delta \models \chi(t_0)$. $t_0$ is the only transition in $\pi$, so $\chi(\pi) = \chi(t_0)$ and we are done.

For the induction step, assume the statement holds for all traces of length $k$. Let $\pi$ be a trace of length $k + 1$, $\pi = t_0, t_1, \ldots, t_k$, where $t_i = \sigma_i \xrightarrow{} \sigma_{i+1}$. Exactly like in the base case, from $\delta \models L(\sigma_{k+1})$ we get $\delta \models L(\sigma_k) \lor \chi(t_k)$. If $\delta \models \chi(t_k)$, then since $\chi(\pi)$ is a disjunction of all $\chi(t_i)$, $\delta \models \chi(\pi)$ as required. Otherwise $\delta \models L(\sigma_k)$. Let $\pi'$ be the prefix $t_0, \ldots, t_{k-1}$. $\pi'$ has $k$ transitions and ends in $\sigma_k$, so by the induction hypothesis, $\delta \models \chi(\pi')$. This means that there is some $t_j$ s.t. $\delta \models \chi(t_j)$. But since $\pi'$ is a subtrace of $\pi$, $t_j$ is in $\pi$, so $\delta \models \chi(\pi)$.

Now, let $\sigma \in \Sigma_P$ be a state such that $\sigma \not\in S$. Assume by contradiction that $\delta \sim \sigma$. Then there exists some trace $\pi$ that ends in $\sigma$ s.t. $\delta \not\models \chi(\pi)$. Since $\delta \models \psi$, $\delta \models \text{av}(\sigma)$. We know the algorithm terminates only when $\text{av}$ is a fixed point of $\text{trans}$. Thus we can apply Lemma 4.1.7, and get that $\delta \models \chi(\pi)$, a contradiction.

**Corollary 4.1.8** Let $\psi$ be as before and $\delta \models \psi$. If a state $\sigma \in \Sigma_P$ is reachable in $\langle \sigma_0, \Sigma_P, T_\delta \rangle$ then $\sigma \models S$.
Proof: Assume $\sigma$ is reachable. Then by Lemma 4.1.5, $\delta \models \sigma$. But by Theorem 4.1.6 this means $\sigma \models S$, a contradiction.

Optimality

In Theorem 4.1.6 we have shown that the constraints given by any fixed point of $\text{trans}$ are sound. That is, if we enforce these constraints, then no error state is reachable. However, this, by itself, is not a particularly interesting property. For example, the labeling $L = \lambda \sigma. \text{ff}$ is also a fixed point of $\text{trans}$. (Unless the set $\{\sigma \mid \sigma \not\models S\}$ is empty this constraint is unenforceable, so soundness holds vacuously, and if it is empty than the program is already correct and any labeling is sound.)

The algorithm we presented above is not only sound, but also maximally permissive. That is, the formula $av(\sigma)$ is the “weakest” (most permissive) formula that describes the constraints that must be enforced to make $\sigma$ unreachable. To prove this we must show the converse of Theorem 4.1.6.

Theorem 4.1.9 Let $\psi$ be as before, and $\delta \subseteq \forall$. If $\forall \sigma \in \Sigma_P. (\sigma \not\models S \implies \delta \not\models \sigma)$, then $\delta \models \psi$.

Informally, this theorem states that if enforcing a set of constraints would make all error states unreachable, then this set of constraints satisfies $\psi$. This means it’s impossible to avoid all error states without satisfying $\psi$, so it is the maximally permissive formula.

Proof: Again, we first prove a more general lemma.

Lemma 4.1.10 Let $L^0 = L_0$, and $L^{k+1} = \text{trans}[L^k]$. For any $k$, and any $\sigma \in \Sigma_P$, if $\delta \not\models L^k(\sigma)$ then there exists a trace $\pi$ from $\sigma_0$ to $\sigma$ s.t. $\delta \not\models \chi(\pi)$.

Proof: We prove by induction on $k$. For the case $k = 0$, if $\sigma \neq \sigma_0$, then $L^0(\sigma) = \text{tt}$, so for any $\delta$, $\delta \models L^k(\sigma)$, and the statement is vacuously true. If $\sigma = \sigma_0$, then the empty trace satisfies the required condition.

For the induction step, assume the statement holds for $L^k$. $\delta \not\models L^{k+1}(\sigma)$. Since $L^{k+1}(\sigma) = L^k(\sigma) \land \text{avoid}(L^k, \sigma)$, we have one of two options:

1. If $\delta \not\models L^k(\sigma)$, then the required trace $\pi$ exists by the induction hypothesis.

2. Otherwise, necessarily $\delta \not\models \text{avoid}(L^k, \sigma)$. So there exists some transition $t = \mu \rightarrow \sigma$ s.t. $\delta \not\models L^k(\mu) \lor \chi(t)$. So, $\delta \not\models \chi(t)$, and since also
δ \not\models L^k(\mu) by the induction hypothesis there exists a trace π' from σ_0 to \mu s.t. δ \not\models \chi(π'). We can now build the trace π to σ by extending π' with the transition t. Since \chi(π) = \chi(π') \lor \chi(t), we have that δ \not\models \chi(π), as required.

Suppose by contradiction that the theorem is false. Then there exists an δ s.t. \forall \sigma \in \Sigma_P. (\sigma \not\models S \implies δ \not\models \sigma) but δ \not\models \psi. Then there is some σ s.t. σ \not\models S and δ \not\models \text{avoid}(\sigma). Since avoid = L^k for some k, we can apply Lemma 4.1.10. By the lemma, there is a trace π from σ s.t. δ \not\models \chi(π). This implies that δ \not\models σ, a contradiction.

\[\Box\]

Syntactic Enforcement in the Program

Theorem 4.1.6 implies that to cut all error states from the transition system we must enforce some satisfying assignment δ of the formula ψ produced by our algorithm. That means we wish to cut from the transition system some set of edges R = T \setminus T_δ. Unfortunately, we cannot operate directly on the transition system. Rather, we must rely on some syntactic mechanism to produce a suitable program P'. In our case, the mechanism is insertion of memory fence instructions. In the next section we explore the translation of constraints into syntactic fences.

4.2 Instantiation for RLX

In the previous section, a general algorithm to infer optimal constraints was shown. When we instantiate this algorithm for a specific memory model, we first need to choose the type of constraints we can actually enforce. That means our constraints must satisfy at least the following basic property: For every transition t to which we assign χ(t) \neq \emptyset it is possible use fences to construct a program P' for which the transition system does not contain t. Note that this is still imprecise, since it requires us to identify the same transition in two different transition systems.

The constraint language we chose for RLX is the set of “ordering constraints” of the form ψ = [l_1 \prec l_2] where l_1, l_2 \in Labs. Intuitively, enforcing the constraint [l_1 \prec l_2] means that P' cannot execute the instruction with label l_2 out of order with respect to the instruction at label l_1. We then
define $\chi(t)$ for a transition $t = \sigma \xrightarrow{l_t} \sigma_2$ to be $\chi(t) = \{[l < l_t] \mid l <_{\sigma,p} l_t\}$. This means that $[l < l_t]$ appears in $\chi(t)$ if:

- The transition was caused by executing the instruction at label $l_t$ by some process $p$.
- The execution buffer $E_\sigma(p)$ contained an instance of the instruction at $l$ before the instruction at $l_t$.

This is equivalent to saying the transition $t$ represents the instruction at $l_t$ being executed while bypassing $l$.

To show the chosen constraint language is indeed useful, we need to show a correspondence between these constraints and syntactic fences. More concretely, we need to show that:

- We know how to enforce any constraint formula produced by the algorithm by adding fences to $P$.
- Enforcement can be done efficiently. To see this is a non-trivial property, consider the constraint language consisting of a single constraint $\beta$, where the enforcement mechanism is “If $\psi = \beta$ add a fence between every two instructions in $P$”. Clearly, we can enforce this constraint, and enforcing it would create a correct program, but this is not the desired outcome.

To achieve this we prove two theorems: Theorem 4.2.3 and Theorem 4.2.5.

First, we show that adding fences can never introduce new error states. It is clear (but slightly tedious to prove) that adding nop operations (with new labels) to a program has no effect on the program’s behavior. So it is enough to show that the set of behaviors a program $P$ with a fence at label $l$ has is a subset of the possible behaviors of $P$ with a nop at the same label. We can show this through a simple simulation argument. The simulation also establishes a correspondence between states (and transitions) in a program $P$ and a program with added fences $P'$ that will be useful throughout the remainder of this chapter.

**Lemma 4.2.1** Let $P$ be a program with a nop instruction at label $l$, and $P'$ the program $P$ with the nop replaced by a fence. Then $\langle \sigma_0, \Sigma_P, T_P \rangle$ simulates $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$.

**Proof:** We define a simulation relation $H$ as follows. Let $\sigma' \in \Sigma_{P'}$. Since $\sigma'$ is reachable, there exists some finite trace $\pi' = \sigma'_0, ..., \sigma'$ in $\langle \sigma_0, \Sigma_P, T_P \rangle$. We can construct a (unique) trace $\pi$ with the final state $\sigma$ in $\langle \sigma_0, \Sigma_P, T_P \rangle$. 

39
by replacing every fence transition with label $l$ with the appropriate nop transition. Let $h(\sigma')$ denote the $\sigma$ derived in this fashion from $\sigma'$. Then $H = \{\sigma', h(\sigma') \mid \sigma' \in \Sigma_P\}$

To show this is indeed a simulation, there are three main things that need to be proven:

1. The trace $\pi$ used in the construction of $H$ is always a legal trace in $\langle \sigma_0, \Sigma_P, T_P \rangle$.
2. If $(\sigma', \sigma) \in H$, then the valuations of all variables ($L$ and $G$) are the same in both.
3. If $(\sigma', \sigma) \in H$ and there is some transition $\sigma' \xrightarrow{t'} \tau' \in T_{P'}$ then there exists a state $\tau \in \Sigma_P$ s.t. $\sigma \xrightarrow{t} \tau \in T_P$ and $(\tau', \tau) \in H$.

For these three conditions, we only give a proof sketch. First, it is easy to show the trace is legal by induction on the number of appearances of $l$ in $\pi'$. The main idea is that replacing a nop by a fence means the buffer $E_{\sigma'}(p)$ is always a subsequence of the buffer $E_{\sigma}(p)$. For the second condition, note that fence is in fact a “nop transition”, and the only effect a fence instruction has is on “bypass” conditions. Thus if the constructed trace $\pi$ is legal, the valuations of variables are exactly the same as in the original trace $\pi'$. Finally, the third conditions holds by construction of $H$: similarly to the first condition, $t'$ must have a corresponding transition $t$ and by using them to extend $\pi'$ and $\pi$ we will get that $(\tau', \tau) \in H$.

This lemma is trivially extendable to replacing any number of nop instructions by fences. Note also that the function $h$ is injective by construction. We can also extend $h$ to transitions, by identifying the transition $h(t')$ where $t' = \sigma'_1 \xrightarrow{t'} \sigma'_2$ with the transition between $h(\sigma'_1)$ and $h(\sigma'_2)$.

Now that we have established inserting fences cannot add new error states, the next thing we wish to show is that we can in fact use syntactic fences to cut any transition $t$ s.t. $\chi(t) \neq \emptyset$. This is established by the following lemma.

**Lemma 4.2.2** Let $P$ be a program, $t = \sigma_1 \xrightarrow{\mathit{lt}} \sigma_2$ a transition in $\langle \sigma_0, \Sigma_P, T_P \rangle$ and $v \in \chi(t)$, where $v = [l < l_t]$. Let $P'$ be the program $P$ modified s.t. a fence instruction is placed on every control path between $l$ and $l_t$. Then there is no $t' \in \langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$ s.t. $h(t') = t$.

**Proof:** Suppose by contradiction there is such a $t' = \sigma'_1 \xrightarrow{t'} \sigma'_2$. Because $v \in \chi(t)$, we know two facts:

40
1. The transition \( t' \) executed the instruction at \( l_t \).

2. Due to the buffer structure, \( v \in \chi(t') \), so there is some process \( p \) s.t. \( l <_{\sigma',p} l_t \). So \( l_t \) bypassed \( l \).

We now need to show that if there is a fence on every control-flow path between \( l_t \) and \( l_1 \) then \( l <_{\sigma',1} p l_t \) implies \( l <_{\sigma',1} p l_f <_{\sigma',1} p l_t \) where \( l_f \) is a fence instruction. This will complete the proof, as it means that \( l_t \) bypassed \( l_f \), but this is impossible according to RLX semantics: Assume this is not the case. Then there is an instance of \( l \) and an instance of \( l_t \) in \( E_{\sigma'}(p) \) with no fence instruction between. There are now two cases:

1. There was never any fence instruction between them in the buffer.
2. There was a fence instruction but it is no longer in the buffer.

However, both cases are impossible. The first because there is a fence on every control path between \( l \) and \( l_t \). We can show by the semantics of BRANCH this implies a fence must be added to the buffer between an addition of \( l \) and an addition of \( l_t \). The second is impossible due to the fact a fence cannot leave a buffer until all previous instructions have left (this is required by all RLX reordering tables).

A corollary of the lemma above is that we can enforce any constraint \( v = [l_1 < l_2] \) (thus cutting any transition \( t \) s.t. \( v \in \chi(t) \)) by placing a fence on every control path between \( l_1 \) and \( l_2 \).

We can now prove the main soundness theorem.

**Theorem 4.2.3** Let \( P \) be a program, \( S \) a specification, \( \psi = \bigwedge \{ av(\sigma) \mid \sigma \not\models S \} \) and \( \delta \models \psi \). Let \( P' \) be the program \( P \) modified s.t. for any \([l_1 < l_2] \in \delta \) a fence instruction is placed on every control path between \( l_1 \) and \( l_2 \). Then \( \forall \sigma \in \Sigma P', \sigma \models S \).

**Proof:** Suppose by contradiction that there is some reachable state \( \sigma' \in \Sigma P' \) s.t. \( \sigma' \not\models S \) (or, equivalently, \( \sigma' \models S \)). By Lemma 4.2.1 there exists some reachable state \( \sigma \in \Sigma P \) s.t. \( \sigma \not\models S \). Since \( \sigma' \) is reachable, there must be a trace to \( \sigma' \) in \( \langle \sigma_0, \Sigma P', T_{P'} \rangle \). Let \( \pi' \) be such a trace. Again, from Lemma 4.2.1 we can construct a trace \( \pi \) in \( \langle \sigma_0, \Sigma P, T_P \rangle \) from \( \sigma_0 \) to \( \sigma \) s.t. for every transition \( t' \) in \( \pi' \) there is a corresponding transition \( h(t') \) in \( \pi \). Since \( \sigma \not\models S \), \( \delta \models av(\sigma) \). From Theorem 4.1.6 we know that \( \delta \not\models \sigma \), and \( \delta \models \chi(\pi) \). So, there is some transition \( t = \sigma_1 \rightarrow \sigma_2 \) in \( \pi \) s.t. \( \delta \models \chi(t) \). This means there is some \( v = [l < l_t] \) s.t. \( v \in \delta \) and \( v \in \chi(t) \). Since \( v \in \delta \), there is a fence on every control path between \( l \) and \( l_t \) in \( P' \). However, since \( v \in \chi(t) \),
by Lemma 4.2.2 there is no transition in \( \langle \sigma_0, \Sigma_p, T_P \rangle \) such that \( h(t') = t \), a contradiction.

Unfortunately, the converse of the soundness theorem does not hold: given a minimal satisfying assignment \( \delta \) of \( \psi \), we are not required to enforce it (by placing fences on every control path for every \( [l_1 < l_2] \in \delta \)) to obtain a correct program. The actual requirement is in fact much weaker.

We say a constraint \( [l_1 < l_2] \) is weakly enforced in a program \( P' \) if a fence is added on some control path between \( l_1 \) and \( l_2 \). We can then show that to obtain a correct program \( P' \), we must weakly enforce some minimal assignment.

**Lemma 4.2.4** Let \( P \) be a program, \( t = \sigma_1 \xrightarrow{l_t} \sigma_2 \) a transition in \( \langle \sigma_0, \Sigma_p, T_P \rangle \). Let \( P' \) be the program \( P \) modified by adding fences, s.t. for every \( v = [l < l_t] \in \chi(t) \) there is no fence instruction on any control path between \( l \) and \( l_t \). If there is in \( \langle \sigma_0, \Sigma_{p'}, T_{P'} \rangle \) a state \( \sigma'_1 \) s.t. \( h(\sigma'_1) = \sigma_1 \) then there is also a transition \( t' \) in \( \langle \sigma_0, \Sigma_{p'}, T_{P'} \rangle \) s.t. \( h(t') = t \).

**Proof:** Let \( \pi' \) be a trace to \( \sigma'_1 \) in \( \langle \sigma_0, \Sigma_{p'}, T_{P'} \rangle \). Then there is a corresponding trace \( \pi \) to \( \sigma_1 \) in \( \langle \sigma_0, \Sigma_p, T_P \rangle \). Let \( p \) be the process which executed the instruction associated with \( l_t \). Since \( t \) is a valid transition in \( \langle \sigma_0, \Sigma_p, T_P \rangle \) we know there is an instance of \( l_t \) in the buffer \( E_{\sigma_1}(p) \) that can bypass any previous instruction in the buffer. We need to show this is also the case for \( E_{\sigma'_1}(p) \). It is clear the two buffers (for \( \sigma_1 \) and \( \sigma'_1 \)) are the same except for, possibly, some added fences. Assume by contradiction there is an added fence at label \( l_f \) in the buffer, s.t. for some \( l, l < \sigma'_1, p l_f < \sigma'_1, p l_t \). Since it must have been added between \( l_1 \) and \( l_t \), that means \( l_f \) lies on some control path between \( l \) and \( l_t \). But since \( [l < l_t] \in \chi(t) \), this is a contradiction. So, a transition \( t' = \sigma'_1 \xrightarrow{l_{t'}} \sigma'_2 \) is legal according to the semantics, and it is easy to show that \( h(\sigma'_2) = \sigma_2 \).

**Theorem 4.2.5** Let \( P \) be a program, \( S \) a specification, \( \psi = \bigwedge \{ av(\sigma) \mid \sigma \notin S \} \), and \( P' \) the program \( P \) modified by inserting fences. If for every satisfying assignment \( \delta \models \psi \) there exists \( [l_1 < l_2] \in \delta \) s.t. there is no fence on any control path between \( l_1 \) and \( l_2 \), then there is some \( \sigma \in \Sigma_{p'} \) s.t. \( \sigma \notin S \).

**Proof:** Let \( \langle \sigma_0, \Sigma_{p'}, T_{P'} \rangle \) be the transition system of \( P' \), and \( E = \{ \sigma \in \Sigma_p \mid \sigma \notin S \} \) the set of all error states in the original transition system.
Assume by contradiction that for all states $\sigma \in \Sigma_{P'}, \sigma \models S$. Then for any $\sigma \in E$, $h(\sigma) \not\in \Sigma_{P'}$. Let $\Pi$ be the (possibly infinite) set of traces in $\langle \sigma_0, \Sigma_P, T_P \rangle$ that lead to states in $E$. Let $\pi$ be some trace in $\Pi$. Since $h(\sigma) \not\in \Sigma_{P'}$, $\pi$ has no equivalent trace in $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$, so some transition $t_\pi$ in $\pi$ has necessarily been cut. By Lemma 4.2.4, this means that there is some constraint $v_\pi = [l_{\pi_1} \prec l_{\pi_2}]$ in $\chi(t_\pi)$ s.t. there is a fence on some control path between $l_{\pi_1}$ and $l_{\pi_2}$. If there are several such constraints, we choose one arbitrarily. We can find such $v_\pi$ for every $\pi \in \Pi$, so let $\delta = \{v_\pi \mid \pi \in \Pi\}$. Now, consider the transition system $\langle \sigma_0, \Sigma_P, T_\delta \rangle$ where $T_\delta = \{t \in T_P \mid \delta \not\models \chi(t)\}$. By construction of $\delta$, $T_\delta \subseteq T'$. This means $T_\delta$ contains no error states, so by Theorem 4.1.9 (maximal permissiveness of $\psi$), $\delta \models \psi$. This means, by the assumption in the theorem, that there exists some $v = [l_1 \prec l_2] \in \delta$ s.t. $P'$ contains no fence on any control path between $l_1$ and $l_2$, but this is a contradiction to the construction of $\delta$.  

**Synthesizing Fences from Constraints**

Theorem 4.2.3, as an extension of Theorem 4.1.6, shows that we can syntactically implement any solution to the constraint formula $\psi$ produced by our algorithm. It shows that if for every constraint $[l_1 \prec l_2]$ that needs to be enforced fences are placed on all control-flow paths between $l_1$ and $l_2$ then the resulting program is safe.

Unfortunately, while Theorem 4.2.5 shows a fence must be placed on some control path between $l_1$ and $l_2$, it does not require placing a fence on all of them. There are, in fact, several reasons a fence placement constructed by simply taking some minimal satisfying assignment $\delta$ of $\psi$ and adding fences on all control-flow paths may be suboptimal:

- First of all, it is not even clear which optimality metric we should use. The number of (static) fences added to the program seems like a convenient choice, but may be misleading. Several fences placed before a loop may have a much smaller (dynamic) execution cost than a single fence placed inside the loop body.

- Theorem 4.2.5 shows we must add a fence on some control path of every constraint that belongs to a minimal satisfying assignment, as opposed to all control-flow paths. This is not a weakness of the theorem, as placing fences on all paths is in fact not always required. This
may happen for two reasons. First, some control-flow paths may be simply infeasible, and putting fences on these paths is clearly unnecessary. More subtly, it is possible that a given re-ordering of instructions is only harmful on some execution paths. Our chosen constraint language does not preserve enough information to make these distinctions. We could use an alternative constraint language to preserve it, but this would dramatically increase the size of $\Lambda_P$ — the number of possible constraints would be exponential in the number of labels, as opposed to merely quadratic.

- Often, it is possible to satisfy several constraints with a single fence. Thus a judicious placement of fences is still required, even once a minimal assignment to the constraint formula is known. Moreover, different minimal assignments may lead to different placement tradeoffs.

We resolve the first issue by working with the natural partial order on fence placements: a set of added fences $C$ is better then a set $C'$ if $C' \subseteq C$. We then produce all minimal incomparable placements and leave the choice between them to the programmer. Choosing between incomparable (by containment) fence placements is a separate hard problem, which we leave to future work.

The second issue could be resolved by adopting a more precise “flow-sensitive” constraint language. This could be done by encoding in the constraints $\chi(t)$ of a transition $t = \sigma_1 \rightarrow \sigma_2$ information about program paths that lead to $t$. Moreover, if we used a “context-sensitive” implementation mechanism instead of fences (for example “conditional fences” — fences that are only sometimes executed, depending on the current program state) we could use even finer constraints. For the input programs we used (cf. Chapter 6), none of these improvements were necessary. Therefore, we also defer examination of these alternatives to the future.

The third issue requires further examination. While there are in general many ways to implement a given constraint $v = [l_1 \prec l_2]$, for simple programs it usually sufficient (while clearly not optimal in general) to consider two options:

- Place a fence immediately after instruction $l_1$
- Place a fence immediately before instruction $l_2$ (if there are branch instructions pointing to $l_2$ they should point to the newly added fence).
This is complicated slightly by the fact that even in this case, there is interdependence between constraints. For example, consider a program with three statements with labels \( l_1, l_2, l_3 \) in sequence and the constraint formula \( v_1 \land v_2 \) where \( v_1 = [l_1 < l_2] \), \( v_2 = [l_1 < l_3] \). Obtaining the (only) solution \( \{v_1, v_2\} \) and then deciding to place a fence immediately before \( l_2 \) (to enforce \( v_1 \)) and before \( l_3 \) (to enforce \( v_2 \)) will result in a placement that contains two fences, instead of the expected single fence after \( l_1 \). We solve this by replacing the constraint formula \( \psi \) with a new formula \( \zeta \).

A fence may only be placed after an existing code label. Therefore, for each label we define a new variable \( v_l \). We also define a function \( p : Labs \rightarrow Labs \) that returns for each label \( l \) the preceding (in the program code) label. We then produce \( \zeta \) by replacing every variable \( v = [l_1 < l_2] \) in \( \psi \) with the clause \( v_{l_1} \lor v_{p(l_2)} \). It is easy to show that every satisfying assignment to \( \zeta \) still produces a sound fence placement. However, it alleviates the interdependence problem by “off-loading” it to a SAT solver. In the preceding example, the formula \( v_1 \land v_2 \) is transformed into \( v_{l_1} \land (v_{l_1} \lor v_{l_2}) \), with the minimal solution \( \{v_{l_1}\} \) as desired.

### 4.3 Limitations

The main drawback of the algorithm described in this section is the fact that it requires explicit enumeration of the program’s state-space. While this is possible for some programs, many programs for which we want to infer fences do not allow such explicit enumeration because the state-space is not finite. This might happen due to a combination of several reasons. Three common reasons are:

1. We are interested in inferring a fence placement for an open system (e.g. library code), and not a single finite-state program.
2. The program for which we wish to infer fences utilizes a potentially unbounded number of heap locations.
3. The program is finite-state under SC but not finite-state under the desired relaxed memory model.

In case (1), the problem boils down to the fact we are not interested in placing fences in a single program. Rather, we want to place fences in the code
of a library implementation such that it remains correct irrespective of the code using the library (the data structure client). A different way to phrase this is to say we want the most general client (which represents all possible clients) of the library to be correct. Unfortunately, the most general client itself is usually not a finite-state program. For example, consider a queue implementation that uses a linked list as the underlying data representation. A client that may add an unbounded number of elements to the queue will use unbounded memory, and the state-space for the client/queue combination is unbounded. In general, this is a hard problem that we do not try to completely solve. We attempt to reduce it using two methods:

- Hand-picked clients that we believe to be representative of the data structure’s behavior.
- Exhaustive enumeration of clients up to a specified bound on the number of operations.

Neither of these two solutions produces a sound verification (or fence inference) procedure. However, as Chapter 6 shows, in practice these methods allow us to infer optimal fences for realistic data structures, as in Tab. 1.1. We have verified that the results are indeed optimal by manually comparing them to fence placements found in the literature.

In case (2), the problem is that due to use of an unbounded number of heap locations, the program is infinite-state even under the sequentially-consistent model. One way to deal with this problem is to “work around it” by applying the algorithm to slightly different programs, and dealing with the difference separately (e.g. using finite-state clients instead of the infinite-state most general client as in case (1)). Another is to use heap abstractions, a subject we will elaborate on in Chapter 8.

In case (3), the problem lies squarely within the focus of our work. As Atig et al. have shown [5], this is in general a very hard problem. Given a finite-state (under SC) program \( P \), deciding reachability for the same program under SPARC TSO or PSO has non-primitive recursive complexity. Further, under SPARC RMO, reachability for SC-finite-state programs becomes undecidable. One way to deal with problems of this kind is through the use of abstract interpretation techniques, and to this we devote the next chapter of this thesis.
Chapter 5

Abstract Memory Models

Abstraction is real, probably more real than nature. – Josef Albers

In Chapter 4 we presented an algorithm for fence inference that works for programs that have a finite state-space under the relaxed memory model. In this chapter, we relax the requirements and only require the program to have a finite state-space under the sequentially consistent model. While this distinction may seem subtle, there are in fact many interesting programs that fall into the “finite-state under SC, infinite-state under RMM” category. A very simple example can be seen in Fig. 5.1.

Under sequential consistency, this program allows a single infinite trace (process 0 runs alone, without process 1 ever taking a step), but this trace only goes through a small number of different states. However, consider the execution of this program under RMO in the RLX framework. In this case, there exists a legal trace in which each iteration of the loop adds all 3 instructions $A_1 - A_3$ to the execution buffer, and then chooses to execute the

\begin{align*}
A_1: &\text{ load } v \leftarrow y \\
A_2: &\text{ store } x \leftarrow 1 \\
A_3: &\text{ if } v == 0 \text{ branch } A_1 \\
B_1: &\text{ store } y \leftarrow 1
\end{align*}

Figure 5.1: Example program with unbounded state-space under relaxed models
newly added branch instruction at $A_3$. This means that after $k$ iterations
the execution buffer $E(0)$ will contain $2k$ instructions, and the state-space
of the program is clearly infinite.

The example above shows the most common pattern that causes SC
finite-state programs to become infinite-state under an RMM. Whenever a
program contains a spin-loop with a potentially unbounded number of itera-
tions, and the loop body does not contain a fence, the amount of information
we must keep track of to preserve the memory model semantics also grows
without a bound. Therefore if we wish to analyze such programs, we must
somehow “collapse” the state-space back to a finite (and manageable) size.
One standard way to do so is to apply abstract interpretation [23]. Before
we go into the details of our technique, we need to present an introduc-
tion to abstract interpretation, and another kind of concrete memory model
semantics.

5.1 Introduction to Abstract Interpretation

A full introduction to abstract interpretation (AI) is well beyond the scope
of this thesis, and can be found in papers by Cousot et al. [23] and a book
by Nielson et al. [59]. Here we introduce a very restricted informal view of
AI that is useful for our purposes.

As shown in Chapter 1, we can define the state-space of a program $\Sigma_P$
as a countable union of transformer applications. For some (“finite-state”) pro-
grams, the result of this union is finite. However, this is not true in
general. This means that techniques relying on state-space enumeration
cannot be used directly.

To solve this problem, we can define a set of abstract states $A$. The
intent is that each abstract state $\sigma \in A$ represents a (possibly infinite) set
of concrete states. We can then use these abstract states to construct an
abstract transition system that is finite, but over-approximates the possible
behaviors of the (infinite) concrete transition system.

Technically, we need to define an abstract domain $A$. This is usually
the powerset $2^A$ or some subset of the powerset that forms a complete join
semi-lattice. Note that much more general abstract domains are possible,
but we do not use them in this thesis. In a common simple case, $A = 2^A$,
and the order relation over $A$ is the standard containment relation.

48
For ease of presentation, throughout this chapter, when both abstract and concrete objects are under discussion we will mark the concrete objects with a ♮ mark. For instance, we refer to the concrete transformer function as τ ♮.

To move between the concrete and abstract domains, an abstraction function \( \alpha : 2^C \rightarrow A \) and a concretization function \( \gamma : A \rightarrow 2^C \) must be defined. Intuitively, for a set \( \Sigma^\flat \subseteq C \) of concrete states, \( \alpha(\Sigma^\flat) \in A \) is a set of abstract states that represents all elements of \( \Sigma \). Conversely, for a set of abstract states \( \Sigma \in A \), \( \gamma(\Sigma) \) is the set of all concrete states represented by \( \Sigma \).

A useful way to think about abstraction and concretization is in terms of information loss. Let \( \Sigma = \alpha(\Sigma^\flat) \). If \( \gamma(\Sigma) = \Sigma^\flat \) then the set of abstract states \( \Sigma \) represents the same information about the program state as \( \Sigma^\flat \). However, in most cases (e.g. if \( 2^C \) is infinite and \( A \) is finite), this is impossible. We therefore only require that \( \Sigma^\flat \subseteq \gamma(\Sigma) \). This means that by applying \( \alpha \) and then \( \gamma \) we have lost information. Originally, we knew the program state is in some concrete set \( \Sigma^\flat \), but after abstraction, we only know that it is in a superset of \( \Sigma^\flat \). To formally capture this intuition we require that \( \alpha \) and \( \gamma \) form a Galois connection.

In this thesis we use a specific type of abstraction functions. We first define a representation function \( \hat{\alpha} \) that translates a single concrete state \( \sigma^\flat \in C \) into the abstract state \( \sigma \in A \) that represents it.

\[
\alpha(\Sigma^\flat) = \bigsqcup_{\sigma^\flat \in \Sigma^\flat} \{\hat{\alpha}(\sigma^\flat)\}
\]

\[
\gamma(\Sigma) = \{\sigma^\flat \mid \{\hat{\alpha}(\sigma^\flat)\} \subseteq \Sigma\}
\]

In addition to the abstract domain we must also define an abstract transformer \( \tau : A \rightarrow A \). For our abstraction to be sound, the abstract transformer must over-approximate the effect of the concrete transformer. That is, we require that for any \( \Sigma^\flat \in C \), we have \( \tau^\flat(\Sigma^\flat) \subseteq \gamma(\tau(\alpha(\Sigma^\flat))) \) or equivalently \( \alpha(\tau^\flat(\Sigma^\flat)) \subseteq \tau(\alpha(\Sigma^\flat)) \). For powerset (and similar) domains, we again define the abstract transformer pointwise. That is, as before, for \( \sigma \in A \), \( \tau(\sigma) \) is defined according to the abstract semantics. For any non-singleton \( \Sigma \in A \), we define \( \tau(\Sigma) = \bigsqcup \{\tau(\sigma) \mid \sigma \in \Sigma\} \).

We can then define abstract transition systems in a fashion similar to...
the concrete case. The abstract state-space is now:

\[ \Sigma^k_P = \tau^k(\sigma_0) \]

\[ \Sigma_P = \bigsqcup_{n \in \mathbb{N}} \Sigma_p^n \]

\[ T_P = \{ \sigma_1 \rightarrow \sigma_2 \mid \sigma_1, \sigma_2 \in \Sigma_P \land \exists \sigma' \in \tau(\{\sigma_1\}).\{\sigma'\} \subseteq \{\sigma_2\} \} \]

Note the definition of \( T_P \) is different from the concrete case, since \( \sigma_1 \in \Sigma_P \) and \( \sigma_2 \in \tau(\{\sigma_1\}) \) no longer imply that \( \sigma_2 \in \Sigma_P \).

It is not hard to show (cf. [59]) that is the abstraction and concretization functions form a Galois connection and the abstract transformer is sound, then we can use the abstract transition system to prove properties of the concrete program. Similarly to the concrete definition \( P \models S \), we can define \( P \models_\alpha S \) that holds if \( \forall \sigma \in \gamma(\Sigma_P).\sigma \models S \), where \( \Sigma_P \) is the abstract transition system under \( \alpha \). Then, under the soundness conditions above, we can show that \( P \models_\alpha S \) implies \( P \models S \).

### 5.2 Partial Store Order

While re-ordering based models are intuitive, they are not very convenient for formal treatment. In this chapter we instead deal with a purely store buffer based model. Unfortunately, RMO cannot be represented using only store buffers. (cf. [49]). Essentially to enable us to analyze a larger class of programs, we restrict ourselves to “stronger” memory models. Thus, the main memory model we deal with in this chapter is “Partial Store Order” (PSO).

The PSO model can be formalized operationally by associating with each processor a set of FIFO queues (store buffers), one for each variable, as shown in Fig. 5.3. The informal semantics of store buffers for PSO can be summarized as follows:

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

The language syntax used here is the same as in Section 3.2. The seman-
tics, however, are not re-ordering based, but rather purely store-buffer
based. This is similar to the definitions in [5, 50, 17].

A (concrete) program state \( \sigma = (G_\sigma, L_\sigma, pc_\sigma, B_\sigma) \in C^\sigma \) is a tuple where:

- \( G \in SVar \) where \( SVar = Shared \rightarrow D \). Valuation of shared variables from the domain \( D \).
- \( L \in Env \) where \( Env = PID \rightarrow (Locals \rightarrow D) \). Valuation of local variables for each process.
- \( pc \in PC \) where \( PC = PID \rightarrow Labs \). Program counters.
- \( B \in SB \) where \( SB \) may differ between different memory models. This is a representation of the store buffers. For PSO, we use \( SB = PID \rightarrow (Shared \rightarrow Seq(D)) \), that is, a separate buffer for any (process, variable) pair.

Above, \( D \) represents the domain from which the variables in the program take values. For example, in the previous chapter, we assumed that \( D = \mathbb{Z} \). Here, we restrict \( D \) to be finite. This is because, further in this chapter, we will require our abstract domain to be finite, which won’t hold using an infinite \( D \). If it is possible for program variables to take an infinite number of values, then an additional “layer” of abstraction is required, that will replace the concrete domain \( D \) with an abstract domain \( D' \), and issue we will discuss further in Chapter 8.

To visualize this structure, consider the program in Fig. 5.2. The program is written in a version of our input language, with structured control flow (while loops) added as syntactic sugar. This program involves two processes that communicate using three shared variables: \( ent0, ent1 \) and \( turn \). A visual representation of the program state appears in Fig. 5.3. Each shared variable has an associated queue which buffers stored values before they are written to main memory. The backward arrows from the buffers to \( P0 \) and \( P1 \) represent the option of load forwarding from the head of the buffer.

In this section, we will often omit \( \sigma \) and usually omit the \( \sharp \) sign, as only concrete states are under discussion. We will also use \( B_p(x) \) as a shorthand for \( B(p)(x) \), when it is clear that \( p \) is the process identifier and not the state.

The semantics of PSO appear in Fig. 5.4. For simplicity we omit the
Process 0:

```
while (true)
{
    store ent0 ← true;
    store turn ← 1;
    do
    
    load e ← ent1;
    load t ← turn;
    }
while (e == true && t == 1);

// Critical Section

store ent0 ← false;
```

Process 1:

```
while (true)
{
    store ent1 ← true;
    store turn ← 0;
    do
    
    load e ← ent0;
    load t ← turn;
    }
while (e == true && t == 0);

// Critical Section

store ent1 ← false;
```

Figure 5.2: Peterson’s Algorithm with explicit memory operations

Figure 5.3: Store buffers for Peterson’s algorithm of Fig. 5.2 under the PSO memory model
\[
\begin{align*}
\text{stmt}(\text{pc}) &= \text{load } r \leftarrow x \quad B(x) = \epsilon \quad G(x) = v \\
L'(r) &= v \quad pc' = \text{next}(pc) \\
\text{(LOAD-G)}
\end{align*}
\]

\[
\begin{align*}
\text{stmt}(\text{pc}) &= \text{load } r \leftarrow x \quad B(x) = b \cdot v \\
L'(r) &= v \quad pc' = \text{next}(pc) \\
\text{(LOAD-B)}
\end{align*}
\]

\[
\begin{align*}
\text{stmt}(\text{pc}) &= \text{store } x \leftarrow r \quad B(x) = b \quad L(r) = v \\
B'(x) &= b \cdot v \quad pc' = \text{next}(pc) \\
\text{(STORE)}
\end{align*}
\]

\[
\begin{align*}
B(x) &= v \cdot b \quad G'(x) = v \\
B'(x) &= b \\
\text{(FLUSH)}
\end{align*}
\]

\[
\begin{align*}
\text{stmt}(\text{pc}) &= \text{fence} \quad \forall x. B(x) = \epsilon \\
pc' = \text{next}(pc) \\
\text{(FENCE)}
\end{align*}
\]

\[
\begin{align*}
\text{stmt}(\text{pc}) &= q \leftarrow \text{cas } x, r, s \\
G(x) &= L(r) \quad L(s) = v \quad B(x) = \epsilon \\
G'(x) &= v \quad L'(q) = \text{true} \quad pc' = \text{next}(pc) \\
\text{(CAS-T)}
\end{align*}
\]

\[
\begin{align*}
\text{stmt}(\text{pc}) &= q \leftarrow \text{cas } x, r, s \\
G(x) &\neq L(r) \quad B(x) = \epsilon \\
L'(q) &= \text{false} \quad pc' = \text{next}(pc) \\
\text{(CAS-F)}
\end{align*}
\]

Figure 5.4: Operational semantics defining a transition from \( \langle G, L, pc, B \rangle \) to \( \langle G', L', pc', B' \rangle \) under PSO
semantics of instructions that do not access shared memory (register operations, branches) and leave expression evaluation implicit. That is, \( L(r) \) is extended to the evaluation of complex expressions \( r \), where a complex expression may only depend on local variables (expression evaluation may not cause a memory access). To avoid clutter, we also omit processor identifiers. A more explicit form of the rules would look more similar to Fig. 3.2. For example, the premise of the FLUSH-\( H \) rule should be read as \( \exists p. H(p)(x) = h \cdot v. \)

Fig. 5.4 shows the role played by the store buffer for storing and loading values to/from main memory. Applying the STORE rule adds a value into a buffer, and applying FLUSH moves a value from a non-empty buffer into main memory. While there are two LOAD rules, when a process performs a load instruction, only one of those rules is enabled. If the appropriate buffer is empty, then LOAD-G is enabled, and the load is performed from main memory. Otherwise, LOAD-B is enabled, and the load is performed from the buffer.

The FENCE and CAS rules have memory fence semantics. These two rules are enabled only when the buffers of the executing process are empty. This means that when a process encounters a fence or CAS instruction, it cannot continue execution until all of the buffers are fully flushed.

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

**Total Store Order (TSO) Model**

A TSO concrete state differs from a PSO concrete state only in the definition of the store buffer. For TSO, there is only a single buffer for all variables of a process. That is, \( SB = PID \rightarrow Seq(Shared \times D) \). The semantics must also be updated to take this difference into account. The flavor of the required changes can be seen in the TSO version of the LOAD-G rule in Fig. 5.5: a non-buffered load requires the entire buffer to be empty, as opposed to a per-variable buffer. Note that as the difference between PSO and TSO lies purely in the grouping of shared variables into store buffers, we can treat them as special cases of the same general model.
The algorithm described in Sec. 4.1 can be directly applied to this memory model. However we must again (like in Section 4.2) instantiate the algorithm for the RMM. In particular, we must define the appropriate set of constraints and the map \( \chi(t) \) that associates constraints with every transition. As we still use syntactic fences to enforce constraints, using the same constraint language as in Section 4.2 seems like the reasonable choice. That is, our constraints are again of the form \( \psi = [l_1 \prec l_2] \) where \( l_1, l_2 \in Labs \). However the definition of \( \chi(t) \) for \( t = \sigma_1 \xrightarrow{t} \sigma_2 \) executed by process \( p \) changes with the semantics. To define it, we must slightly augment the semantics shown in the previous section, as they do not preserve enough information to enable constraint generation. Intuitively, a fence between \( l_1 \) and \( l_2 \) would forbid executions in which \( l_2 \) is executed while a value stored by \( l_1 \) is still in the buffer. However, if we examine the transition \( t = \sigma_1 \xrightarrow{t} \sigma_2 \) and the source state \( \sigma_1 \), we cannot know whether a store by some \( l \) is in the buffer, since buffers only contain the stored value. Therefore we instrument our semantics with additional information on which instruction stored each value. The modified store buffers \( B_p(x) \) contain pairs of the form \( \langle l, v \rangle \) where \( l \in Labs, v \in D \), and the semantics are “tweaked” to support this change. In the remainder of the chapter we will sometimes, for convenience, use uninstrumented semantics in examples and proofs, but the implicit assumption is that the instrumentation is always present.

Using the instrumented buffers, we can define \( \chi(t) \) according to the intuition given above:

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

Where \( p \) is the process that executed the operation represented by \( t \).

We now need to prove the analogue of Theorem 4.2.3 — that is, we must show that we can enforce such constraints using static fences in our
new setting. Examining the proof of Theorem 4.2.3, we can see that it is in fact valid as is, if we prove the analogues of Lemma 4.2.1 (that establishes a simulation of \(\langle \sigma_0, \Sigma_P, T_P \rangle\) by \(\langle \sigma_0, \Sigma_P, T_P' \rangle\) where \(P'\) is \(P\) augmented with fences) and Lemma 4.2.2 (that shows that we are able to enforce every individual constraint). The construction of the simulation is very similar to the RLX case, so we only show the proof of the latter lemma.

**Lemma 5.3.1** Let \(P\) be a program, \(t = \sigma_1 \xrightarrow{l} \sigma_2\) a transition in \(\langle \sigma_0, \Sigma_P, T_P \rangle\) and \(v \in \chi(t)\), where \(v = [l < l_t]\). Let \(P'\) be the program \(P\) modified s.t. a fence instruction is placed on every control path between \(l\) and \(l_t\). Then there is no \(t'\) in \(\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle\) s.t. \(h(t') = t\).

**Proof:** Suppose by contradiction there is such a \(t' = \sigma'_1 \rightarrow \sigma'_2\). Because \(v \in \chi(t)\), we know:

1. The transition \(t'\) executed the instruction at \(l_t\).
2. \(v \in \chi(t')\), so there is some process \(p\), variable \(x\) and value \(v\) s.t. \(\langle l_1, v \rangle \in \text{Set}(B_{\sigma'_1, p}(x))\).

This means there is some trace \(\pi\) that passed through \(l\) and \(l_t\) without the value \(v\) being flushed between the last appearance of \(l\) and \(l_t\). However, since there is a fence on every control path between \(l\) and \(l_t\), some fence must appear in \(\pi\) between those two instructions. This means \(v\) must have been flushed out before \(l\) was encountered, a contradiction. \(\square\)

The proof of the analogue of Theorem 4.2.5 is also similar.

### 5.4 Partial Coherence Abstractions

As noted in the beginning of this chapter, we wish to describe the behavior of programs with possibly unbounded state-space using a bounded representation. To illustrate the kind of behavior we can expect, we again examine the program in Fig. 5.1. Consider an infinite trace \(\pi\) in which process 0 runs continuously without process 1 making an intervening step. Furthermore, in this trace, while there are always enabled FLUSH transitions, those transitions are never taken, and the process always “chooses” to take the transition indicated by \(\text{stmt}(pc)\). After \(k\) iterations of the loop, the store buffer \(B_0(x)\) will contain \(k\) copies of the value 1, so the explored state-space is, again, infinite.
Abstract memory models are our abstract interpretation-based solution to this problem. As demonstrated above, under buffer-based models, the factor that causes a program with finite (under SC) state-space to become infinite-space is the fact buffers can grow without a bound. To solve this we represent the infinite collection of possible buffers of unbounded length using a bounded set of abstract buffers. Every abstract buffer can represent of a (possibly infinite) number of concrete store buffers. An abstract memory model is simply abstract semantics for our programming language in which concrete store buffers are replaced with such an abstract representation. In this section we describe a particular family of abstract memory models based on “partial coherence”.

Abstract Domain

Our goal is to define an abstract memory model that over-approximates PSO. The first question we must deal with is “which information may we abstract away, and what must we preserve in the abstraction?”. By examining concrete executions under concrete PSO semantics we can see they have the following three properties:

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

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

The challenge in designing an abstraction for store-buffer based memory models lies in preserving properties 1-3 (to the greatest possible extent) using a bounded representation for each buffer. To preserve intra-process
coherence, our abstractions maintain “recency” information per variable. To preserve inter-process coherence, our abstractions preserve order between stores up to some constant bound $k$ (a parameter of our abstraction), and treat the remaining stores as an unordered set. While property 2 is not fully preserved, we do preserve partial coherence. The intuition is that if a process stores many (possibly different) values to the same shared variable without an intervening fence, the order in which they become visible may not be important for the correctness of the algorithm. Therefore we allow our abstract model to lose order information between items once the buffer size reaches a certain bound $k$.

Technically, this is achieved by representing a (possibly infinite) set of concrete store buffers as an “abstract buffer”: a tuple $\langle l, S, H \rangle$ where $l$ is a single value, $S$ a set of values, and $H$ a sequence of values with length up to some length $k$. The $l$ element records the last value that was written into the concrete buffer thus retaining “recency” information, which we discuss later. $H$ records the $k$ oldest values in the buffer, in their original order, if these are known. Finally, $S$ records the set of values written into the buffer since the last flush but abstracts away the order as well as the number of times each of these values had been written.

More formally, we can define the function $\hat{\alpha}_B$ in the following fashion. Let $k \in \mathbb{N}$ be some constant, $D$ the domain from which values are taken, and $Q_k = D \times 2^D \times \text{Seq}_{\leq k}(D)$.

Then $\hat{\alpha}_B : \text{Seq}(D) \rightarrow Q_k$ is defined as:

$$\hat{\alpha}_B(B) = \langle \text{last}(B^2), \text{UTail}(B^2, k), \text{head}(B^2, k) \rangle$$

The way to visualize the effect of $\hat{\alpha}_B$ is shown in Fig. 5.6. If the concrete buffer is of length at most $k$, it is simply copied into the $H$ part of the tuple. Otherwise, the leftmost $k$ elements are copied into $H$, while the rest are collected into the set $S$. In both cases, $l$ records the rightmost value. Note that $l$ is redundant for the $|B| < k$ case, and is kept for the sake of consistency.

Similarly, we define the concretization function $\hat{\gamma}_B : Q_k \rightarrow \text{Seq}(D)$.

First, let us define $E\text{Seq}(S) = \{Q \mid Q \in \text{Seq}(S) \land \forall S' \subseteq S.Q \notin \text{Seq}(S')\}$. That is, $E\text{Seq}(S)$ are all finite sequences of elements of $S$ that use every
(a) The length of the buffer is higher than \( k \)  
(b) The length of the buffer is at most \( k \)

Figure 5.6: Abstraction of a single buffer, parameterized by \( k \)

Intuitively, \( \hat{\gamma}_B(B) \) is the set of all concrete buffers represented by \( B = (l, S, H) \). Note that it is quite possible for the same concrete buffer \( B \) to be represented by two different abstract buffers \( B_1, B_2 \). For example, consider \( B = "abc" \) (with \( D = \{a, b, c\}, k \geq 3 \)). Clearly, \( B \in \hat{\gamma}_B(\langle c, \emptyset, "abc" \rangle) \).

However, it is also true that \( B \in \hat{\gamma}_B(\langle c, \{a, b, c\}, e \rangle) \). In fact, it is easy to see that \( \hat{\gamma}_B(\langle c, \emptyset, "abc" \rangle) \subset \hat{\gamma}_B(\langle c, \{a, b, c\}, e \rangle) \). This property naturally leads to the definition of an order relation \( \sqsubseteq_b : Q_k \times Q_k \), where \( B_1 \sqsubseteq_b B_2 \) if \( \hat{\gamma}_B(B_1) \subset \hat{\gamma}_B(B_2) \). Intuitively, \( B_1 \sqsubseteq_b B_2 \) means that \( B_2 \) represents more concrete buffers than \( B_1 \). This relation can be written out more explicitly:

**Lemma 5.4.1** \( B_1 = \langle l_1, S_1, H_1 \rangle \sqsubseteq_b \langle l_2, S_2, H_2 \rangle = B_2 \) if and only if

\[ l_1 = l_2 \]

and

\[ \exists t. H_2 = head(H_1, t) \land S_2 = S_1 \cup UTail(H_1, t) \]

Less formally, this means that \( B_1 \sqsubseteq_b B_2 \) if and only if their most recent elements are equal, and \( B_2 \) can be produced from \( B_1 \) by removing part of the tail end of \( H_1 \) and adding the removed elements into \( S_1 \).

**Proof:** Assume \( l_1 = l_2 \), and \( \exists t. H_2 = head(H_1, t) \land S_2 = S_1 \cup UTail(H_1, t) \).

Let \( B^2 \in \hat{\gamma}(B_1) \). Then \( B^2 = X \cdot H_1 \) where \( X \in ESeq(S_1) \) and \( last(X \cdot H_1) = l_1 \). Let \( t \) be as above, and \( Q = tail(H_1, |H_1| - t) \). Since we know that \( H_1 = Q \cdot H_2 \), we have that \( B^2 = X \cdot Q \cdot H_2 \). It remains to be shown that \( X \cdot Q \in ESeq(S_2) \). First, since \( X \in ESeq(S_1) \) every element of \( X \) appears in
Similarly, all elements of $Q$ appear in $UTail(H_1,t)$. On the other hand, if an element $e$ is in $S_2$, it is either in $S_1$ or $UTail(H_1,t)$. If it is in $S_1$ then since $X \in ESeq(S_1)$ it appears at least once in $X \cdot Q$. If $e \in UTail(H_1,t)$, then by definition of $UTail$, it appears somewhere in $tail(H_1,\lfloor H_1 \rfloor - t) = Q$.

The other direction of the proof is similar. 

The requirement of $ESeq(S)$ that every member of $S$ appear at least once in the sequence means the concrete empty buffer $B^\sharp = \epsilon$ is represented by exactly one abstract buffer $B_\epsilon = \langle \bot, \emptyset, \epsilon \rangle$. Consider some abstract buffer $B = \langle I, S, H \rangle$. If $H \neq \epsilon$, then every concrete buffer in $\hat{\gamma}(B)$ must start with $H$, so it cannot be $\epsilon$. Otherwise, if $H = \epsilon$, but $S \neq \emptyset$ then every $B^\sharp \in \hat{\gamma}(B)$ must be of length at least $\lfloor S \rfloor$, but $\lfloor \epsilon \rfloor = 0$.

Now, we can define the abstract domain of interest $\mathbb{A}$. An abstract state $\sigma = \langle G_\sigma, L_\sigma, pc_\sigma, B_\sigma \rangle \in A$ only differs from a concrete state in the definition of $SB$ (the range of $B_\sigma(p)$). For concrete PSO, we had $SB^\sharp = PID \rightarrow (Shared \rightarrow Seq(D))$. For the abstract partial-coherence domain with bound $k$, we replace those concrete buffers with their abstract counterparts, that is $SB_k = PID \rightarrow (Shared \rightarrow Q_k)$. We can now extend the order $\sqsubseteq_b$ from abstract buffers to abstract states. We define the relation $\sqsubseteq_s$ on $A \times A$ as $\sigma_1 = \langle G_1, L_1, pc_1, B_1 \rangle \sqsubseteq_s \langle G_2, L_2, pc_2, B_2 \rangle = \sigma_2$ if they coincide on $G, L, pc$ and $\forall p, x.B_1(p)(x) \sqsubseteq_b B_2(p)(x)$. Similarly, we can extend $\hat{\alpha}_B$ to a function over states $\hat{\alpha} : C \rightarrow A$.

Finally, we define the abstract domain $\mathbb{A}$ as the set of all antichains of $A$ with respect to $\sqsubseteq_s$:

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

Note that the set of antichains of a partially-ordered set is a complete lattice with the pointwise partial order $\sqsubseteq : \mathbb{A} \times \mathbb{A}$ defined as

$$\Sigma_1 \sqsubseteq \Sigma_2 \text{ iff } \forall \sigma_1 \in \Sigma_1 \exists \sigma_2 \in \Sigma_2. \sigma_1 \not\sqsubseteq_s \sigma_2$$
The join operator over $A$ is defined in the standard manner, as the set of maximal elements of the union

$$\Sigma_1 \cup \Sigma_2 = \{ \sigma \in \Sigma_1 \cup \Sigma_2 \mid \forall \rho \in \Sigma_1 \cup \Sigma_2. \sigma \neq \rho \implies \sigma \not\sqsubseteq \rho \}$$

Our abstraction function $\alpha : C \to A$ (where $C$ is simply $2^C$, the powerset of $C$) is then

$$\alpha(\Sigma) = \bigcup\{\hat{\alpha}(\sigma^\natural) \mid \sigma^\natural \in \Sigma \}$$

The appropriate concretization function $\gamma : A \to C$ is:

$$\gamma(\Sigma) = \{ \sigma \in C \mid \{\hat{\alpha}(\sigma^\natural)\} \sqsubseteq \Sigma \}$$

Note that $\alpha, \gamma$ constructed in this manner form a Galois connection (cf. Nielson et al. [59]).

We used the antichain construction (as opposed to the more natural $A = 2^A$) to avoid situations in which two abstract state-sets represent the same sets of concrete states. E.g, if $\sigma_1 \sqsubseteq s \sigma_2$ and $\Sigma_1 = \{\sigma_1, \sigma_2\}$, $\Sigma_2 = \{\sigma_2\}$, then $\gamma(\Sigma_1) = \gamma(\Sigma_2)$ even though $\Sigma_1 \neq \Sigma_2$, which is technically undesirable.

**Domain Design**

Previously, we presented the details of our abstract domain. As given, this choice of abstract domain may seem somewhat arbitrary. Here we present the rationale for this particular choice.

One of our observations is that in a correct program a process rarely performs a large (or unbounded) number of stores to a single memory location without an intervening fence. This is because programmers do not normally think in terms of unordered stores. So we expect correct programs to only utilize very short buffers. This is validated by our results (cf. Chapter 6) — correct versions of the benchmark algorithms could be verified using $k = 1$. There are, of course, examples of correct programs where buffers may become very long. One well-known example is the Sieve of Eratosthenes implementation in [9]. However, in this case, the same value is written, so the order of writes is irrelevant. The only requirement is that values that were never written do not appear in the buffer “out of thin air”.

Following this observation, our abstraction is designed to: (i) preserve the order between a small number of stores to the same memory location;
abstract away the order in long sequences of stores but preserve the stored values, so that values do not appear “out of thin air”.

A naive approach that stems from the above observation is to bound the buffers to a constant length. However, we wish our verification procedure to remain sound for arbitrary SC-finite-state programs. That is clearly impossible using bounded buffers, as it is trivial to construct an incorrect program which would appear to be correct using bounded buffers of some given length \( k \). Similarly, when the abstraction is used for fence inference, we wish to always infer a correct placement. Therefore we combine a bounded buffer with an unordered set that provides an over-approximation “safety net”.

The practical effect of this over-approximation is that the inter-process coherence requirement is only partially preserved. For example, suppose processor \( p \) stores the values \( a \) and then \( b \) to the variable \( x \). The resulting concrete buffer \( B^x_p(x) \) is “\( ab \)”. Taking \( k = 0 \), the abstract buffer is \( B = \hat{\alpha}_B(“ab”) = \langle b, \{a, b\}, \epsilon \rangle \). Note that, for example, \( \hat{\alpha}_B(“ab”) = \hat{\alpha}_B(“abb”) \).

So given the abstract buffer \( B \) we must allow another process \( q \) to observe the values being written in the opposite order. Worse, since, for example, \( \hat{\alpha}_B(“ab”) = \hat{\alpha}_B(“abab”) \) it is possible for process \( q \) to observe the stores in the opposite order, and for a third process \( r \) to observe them in the original order. Of course, this behavior occurs only if a process performs more than \( k \) stores to the same memory location without an intervening fence or CAS. As long as at most \( k \) stores to a memory location are performed without an intervening fence, the abstract domain is fully precise.

The last element of our buffer abstraction is the \( l \) element that records recency information. This is motivated by the need to preserve the intra-process coherence requirement: a process storing several values to a shared variable \( x \), and then performing a load from \( x \), should not see any value it has itself stored except the most recent one. This is an extremely basic property and an abstraction that does not preserve this information will fail to verify many reasonable programs.

An additional motivation for the design of this domain was recoverability of sequential consistency using fences. Let \( P \) be a program that is correct under SC for which we wish to infer fences under some memory model \( M \). As we noted in the previous chapter for concrete weak memory model the fence inference problem always has a trivial solution — a fence between every two memory access instructions. We call memory models that have this property
SC- Recoverable (SCR). The SCR property may seem trivial, however it is in fact easy to design (abstract) weak memory models for which it does not hold. For instance, as demonstrated in Chapter 6, the partial-coherence abstraction with \( k = 0 \) is not SCR. It is easy to see, however, that for \( k \geq 1 \) partial-coherence abstractions guarantee SCR. Given a program, if a fence is placed immediately after every store instruction, then i) \(|H|\) can never grow above 1 so stores cannot become visible out of order and ii) the stored value can never be observed by the process itself before it is flushed. In effect this makes the store and flush operations atomic, reducing the program’s behaviors precisely to those possible under sequential consistency.

Abstract Semantics

In addition to the abstract domain, we also need to define abstract semantics for the domain, from which we can construct the abstract transformer. The semantics are shown in Fig. 5.7. In the figure, we use the shorthand \( emp(x) \) to mean that \( H(x) = \epsilon \land S(x) = \emptyset \), and the short hand \( upl(x) \) for:

\[
upl(x) = \begin{cases} 
\bot & \text{if } H'(x) = \epsilon \land S'(x) = \emptyset \\
L(x) & \text{otherwise}
\end{cases}
\]

These semantics are a natural modification of the concrete semantics to fit the abstract domain. In the concrete semantics, a process may load the latest value it wrote by reading its own store buffer (rule \( \text{LOAD-B} \)). Correspondingly, in the abstract semantics, the rule \( \text{LOAD-B} \) reads the most recent value recorded in \( l \). Had we not recorded the most recent value \( l \) that a process wrote, a process \( p \) that performs a load when \( S_p(x) \neq \emptyset \) would have to conservatively explore all possible values in the set \( S_p(x) \). The rule \( \text{LOAD-G} \) is similar to the corresponding rule in the concrete semantics: when the buffer is known to be empty, the value is loaded from global store.

The abstract version of the store rules is somewhat more different from the concrete one. In the concrete semantics, a single \( \text{STORE} \) rule added a value to the buffer. In the abstract version, the store semantics are split into two cases — \( \text{STORE-H} \) and \( \text{STORE-S} \) — depending on whether the size of the buffer is known to be of length at most \( k \).

As long as \( S(x) = \emptyset \) and \(|H(x)| < k\), the contents of the buffer are known precisely. Thus, similarly to the concrete \( \text{STORE} \) rule, the effect of a
\[
\begin{align*}
\text{stmt}(pc) & = \text{load } r \leftarrow x \quad \text{emp}(x) \quad v = G(x) \quad \text{(LOAD-G)} \\
L'(r) & = v \quad pc' = \text{next}(pc) \\
\text{stmt}(pc) & = \text{load } r \leftarrow x \quad \neg \text{emp}(x) \quad v = l(x) \quad \text{(LOAD-B)} \\
L'(r) & = v \quad pc' = \text{next}(pc) \\
S(x) & = \emptyset \quad H(x) = h \quad |h| < k \quad L(r) = v \quad \text{(STORE-H)} \\
H'(x) & = h \cdot v \quad l'(x) = v \quad pc' = \text{next}(pc) \\
\text{stmt}(pc) & = \text{store } x \leftarrow r \quad \text{emp}(x) \quad L(r) = v \quad \text{(STORE-S)} \\
S'(x) & = s \cup \{v\} \quad l'(x) = v \quad pc' = \text{next}(pc) \\
H(x) & = v \cdot h \quad G'(x) = v \quad l'(x) = \text{upl}(x) \quad \text{(FLUSH-H)} \\
H(x) & = \epsilon \quad v \in S(x) \quad G'(x) = v \quad \text{(FLUSH-SN)} \\
G'(x) & = v \quad S'(x) = S(x) \setminus v \quad l'(x) = \text{upl}(x) \quad \text{(FLUSH-SD)} \\
\text{stmt}(pc) & = \text{fence} \quad \forall x \in \text{Shared.emp(x)} \quad pc' = \text{next}(pc) \quad \text{(FENCE)} \\
\text{stmt}(pc) & = q \leftarrow \text{cas } x, r, s \quad \text{emp}(x) \quad L(r) = G(x) \quad L(s) = v \quad \text{(CAS-T)} \\
G'(x) & = v \quad L'(q) = \text{true} \quad pc' = \text{next}(pc) \\
\text{stmt}(pc) & = q \leftarrow \text{cas } x, r, s \quad \neg \text{emp}(x) \quad L(r) \neq G(x) \quad \text{(CAS-F)} \\
L'(q) & = \text{false} \quad pc' = \text{next}(pc)
\end{align*}
\]

Figure 5.7: Abstract semantics defining a transition from \(\langle G, L, pc, B \rangle\) to \(\langle G', L', pc', B' \rangle\)
store follows STORE-H, adding the value to the tail of \( H(x) \). When \(|H(x)| = k\), the contents are still known, but the maximal size of \( H \) has been reached and no more values can be added to it. Therefore, the new value is stored in the (unordered) set of values \( S(x) \) (as shown in the rule STORE-S). Finally, if \( S(x) \neq \emptyset \) we have already lost the information on the precise number of elements on the buffer. This means we may not added the value to \( H(x) \) even if it is shorter than \( k \), and are forced to keep updating the set. In all cases, the most recent value \( l(x) \) is updated to the value just stored.

The single concrete FLUSH rule is again split, this time into three cases: FLUSH-H, FLUSH-SN and FLUSH-SD. When the head of the buffer is non-empty (\( H(x) \neq \epsilon \)), we know precisely what the oldest element still in the buffer is. In this case FLUSH-H behaves exactly as the FLUSH rule in the concrete semantics: it selects the oldest element in \( H(x) \), writes it to \( G(x) \) and updates \( H(x) \).

When \( H(x) = \epsilon \) and \( S(x) \neq \emptyset \), any of the values in \( S(x) \) become possible candidates for flushing (since \( S(x) \) is unordered, we do not know which value is the oldest one). The two rules FLUSH-SD (flush from set, destructive) and FLUSH-SN (flush from set, non-destructive) enabled in this case differ on whether the value selected to be flushed is removed from \( S(x) \) or kept in it. This is required since we do not know how many times a value appears in the buffer. Thus, in the concrete domain, FLUSH-SD of a value \( v \) represents a flush of the last instance of \( v \). In contrast, FLUSH-SN represents the situation in which more instances of \( v \) remain in the buffer. As the flush rules affect the head of the buffer, and not the tail, \( l(x) \) must change only if the last element in the buffer was flushed. In this case, the resulting buffer is empty and \( l'(x) \) must be set to \( \perp \). This applies only to the FLUSH-H and FLUSH-SD rules, as applying the FLUSH-SN rule never results in an empty buffer.

**Soundness**

In the previous subsection, we have provided some informal motivation for the abstract semantics. It remains to show that they are indeed formally sound with respect to concrete PSO. The first thing we show is that the semantics are locally sound. This means that for each transition legal in some concrete transition system, there exists an appropriate abstract transition that over-approximates its effect. This is formally stated by Lemma 5.4.2.
Figure 5.8: Diagram showing relationships in Lemma 5.4.2

The lemma statement might appear cumbersome, but in fact states a relatively simple relationship between states that can be visualized by the diagram shown in Fig. 5.8.

**Lemma 5.4.2** Let $\omega^\natural, \sigma^\natural$ be concrete states such that $\sigma^\natural \in \tau^\natural(\omega^\natural)$. Let $\sigma = \hat{\alpha}(\sigma^\natural)$. Let $\omega = \hat{\alpha}(\omega^\natural)$, and $\nu$ some abstract state s.t. $\omega \sqsubseteq_\nu \nu$. Then there exists $\rho \in \tau(\nu)$ s.t. $\sigma \sqsubseteq_\rho \nu$.

**Proof:** Let $\omega^\natural = \langle G_\omega, L_\omega, pc_\omega, B_\omega^\natural \rangle$. By definition of $\hat{\alpha}$, $\omega = \langle G_\omega, L_\omega, pc_\omega, B_\omega \rangle$ where $B_\omega = \lambda p, x.\hat{\alpha}_B(B(p)(x))$. Since $\sqsubseteq_\nu$ requires states to coincide on $G, L, pc$, we have $\nu = \langle G_\omega, L_\omega, pc_\omega, B_\nu \rangle$, s.t. for any $p, x$, we have $B_\omega(p)(x) \sqsubseteq_b B_\nu(p)(x)$. The notation for the expansion of $\sigma^\natural$ and $\sigma$ is similar.

Since $\sigma^\natural \in \tau^\natural(\omega^\natural)$, there is some inference rule that allows the transition from $\omega^\natural$ to $\sigma^\natural$. We establish the existence of $\rho$ on a case-by-case basis depending on the appropriate inference rule. We will write out the first case fully, and skip some trivial parts for the other cases.

- The rule LOAD-G was applied to $\omega^\natural$ for process $p$, variable $x$, register $r$, value $v$: We will obtain $\rho$ by applying (abstract) LOAD-G to $\nu$. First, LOAD-G is enabled for $\nu$ with the same $p, x, r, v$:
  - Since $\hat{\alpha}$ does not change $G$ and $pc$, and LOAD-G was enabled for $\omega^\natural$, stmt$(pc(p)) =$ load $x, r$ and $G(p)(x) = v$. 

66
Again, since $\text{LOAD-G}$ was enabled for $\omega^\flat$, $B^\flat_\omega(p)(x) = \epsilon$. So $B_\omega(p)(x) = \hat{\alpha}_B(\epsilon) = (\bot, \emptyset, \epsilon)$. As shown earlier, for the empty abstract buffer $B_\omega(p)(x) \sqsubseteq_s B_\nu(p)(x)$ implies that $B_\nu(p)(x) = B_\omega(p)(x)$. So $emp_\nu(p)(x)$ also holds.

Now, let us compare $\sigma$ and $\rho$. Since concrete $\text{LOAD-G}$ does not alter the $L$ component, $L_\sigma = L_\omega$. Abstract $\text{LOAD-G}$ does not alter $L$ either, so $L_\rho = L_\nu = L_\omega$. Thus, $L_\rho = L_\sigma$ as required. In the case of $G$, the reasoning is similar, but with $G(p)(x)$ being set to the same value in both cases. That is, $G_\sigma = G_\omega[x \mapsto v]$, and $G_\rho = G_\nu[x \mapsto v] = G_\omega[x \mapsto v] = G_\sigma$. Similar reasoning applies to the $pc$ component.

Finally, neither version of $\text{LOAD-G}$ alters $B$. Since concrete $\text{LOAD-G}$ does not alter $B$, $B^\flat_\omega = B^\flat_\nu$ which implies $B_\sigma = B_\omega$. By our definition of $\nu$, $\forall p, x. B_\omega(p)(x) \sqsubseteq_b B_\nu(p)(x)$. Since abstract $\text{LOAD-G}$ does not alter $B$, $B_\rho = B_\nu$. So, overall, $\forall p, x. B_\sigma(p)(x) \sqsubseteq_b B_\rho(p)(x)$, and this completes the proof that $\sigma \sqsubseteq_s \rho$. In further cases, we will only show that $B_\sigma(p)(x) \sqsubseteq_b B_\rho(p)(x)$, as proof of the rest is either trivial or very similar to this case.

- The rule $\text{LOAD-B}$ was applied to $\omega^\flat$ for $p, x, r, v$. We will obtain $\rho$ by applying (abstract) $\text{LOAD-B}$ to $\nu$. $\text{LOAD-B}$ is enabled for $\nu$ with the same $p, x, r, v$:

  $B_\omega(x) = \hat{\alpha}_B(B^\flat_\omega(x))$. Since $B^\flat_\omega(x) = b \cdot v$, $l_\omega(x) = v$, and $emp_\omega(x)$ does not hold. As before, implies that $l_\nu(x) = v$ and $emp_\nu(x)$ does not hold.

The proof that $\sigma \sqsubseteq_s \rho$ is very similar to the previous case with the roles of $G$ and $L$ exchanged.

- The rule $\text{FENCE}$ was applied to $\omega^\flat$. We will obtain $\rho$ by applying (abstract) $\text{FENCE}$ to $\nu$. $\forall x. B^\flat_\omega(x) = \epsilon$. So $\forall x. B_\nu(x) = B_\omega(x) = (\bot, \emptyset, \epsilon)$, and $\forall x. emp_\nu(x)$. Thus, $\text{FENCE}$ is enabled. Since $\text{FENCE}$ has no effect on any of the state components, $\rho = \omega$.

- The rule $\text{STORE}$ was applied to $\omega^\flat$. Let $B^\flat = B^\flat_\omega(x)$, and $B = (l, S, H) = B_\nu(x)$. We know that $\hat{\alpha}(B^\flat) = B_\omega \sqsubseteq_b B_\nu$. This implies that $B^\flat = X \cdot H$ s.t. $X \in ESeq(S)$ and $\text{last}(B^\flat) = l$. Which rule will be applied to $\nu$ to obtain $\rho$ depends on $S$ and $H$.

67
If \( S = \emptyset \) and \(|H| < k\), we will apply the rule \( \text{STORE-H} \). Clearly, all of the conditions required for \( \text{STORE-H} \) to be enabled for \( \nu \) hold.

Since \( S = \emptyset \), necessarily \( X = \emptyset \) and \( H = B^2 \), so \( B_\nu(x) = \langle l, \emptyset, B^2 \rangle \).

So, by applying \( \text{STORE-H} \), we get \( B_\rho = \langle v, \emptyset, B^2 \cdot v \rangle \). On the other hand, \( B_\sigma^2(x) = B^2 \cdot v \). Since \(|B^2| < k\), \( B_\sigma^2(x) \leq k \), so \( B_\sigma(x) = \langle v, \emptyset, B^2 \cdot v \rangle \), which means \( B_\sigma(x) = B_\rho(x) \).

Otherwise (if \( S \neq \emptyset \) or \(|H| \geq k\)) we will apply the rule \( \text{STORE-S} \). Again, it is clear that all the conditions required for \( \text{STORE-S} \) to be enabled hold. From the abstract semantics, we know that \( B_\rho = \langle v, S \cup \{v\}, H \rangle \), and, as before, \( B_\sigma^2(x) = B^2 \cdot v \). Since we no longer have an upper bound on \(|B^2|\), we require another case-split.

* In the first case, \(|B^2| < k\). So \( B_\omega(x) = \langle l, \emptyset, B^2 \rangle \) and \( B_\sigma(x) = \langle v, \emptyset, B^2 \cdot v \rangle \). Since \( B_\omega \sqsubseteq B_\nu \), there exists some \( t \) s.t. \( S = \text{UTail}(B^2, t) \) and \( H = \text{head}(B^2, t) \). It is easy to see that for the same \( t \), \( S_\rho = S \cup \{v\} = \text{UTail}(B^2 \cdot v, t) \) and \( H = \text{head}(B^2 \cdot v, t) \), so \( B_\sigma \sqsubseteq B_\rho \).

* In the second case, \(|B^2| \geq k\). So, \( B_\omega(x) = \langle l, S_\omega, H_\omega \rangle \), where \( H_\omega = \text{head}(B^2, k) \) and \( S_\omega \) contains the remaining elements (possibly \( S_\omega = \emptyset \)). This implies that \(|B^2 \cdot v| > k\), so necessarily \( v \in \text{UTail}(B^2 \cdot v, k) \). This means that \( B_\sigma(x) = \langle v, S_\omega \cup \{v\}, H_\omega \rangle \). Since \( B_\omega(x) \sqsubseteq B_\nu(x) \), there exists some \( t \) s.t. \( S = S_\omega \cup \text{UTail}(H_\omega, t) \), and \( H = \text{head}(H_\omega, t) \). As before, for the same \( t \), we will get that \( S_\rho = S \cup \{v\} = S_\omega \cup \{v\} \cup \text{UTail}(H_\omega, t) \), and again, we are done.

The rule \( \text{FLUSH} \) was applied to \( \omega^2 \). We now again decide on which abstract rule to invoke according to the contents of \( B^2 \) and \( B_\nu(x) = \langle l, S, H \rangle \).

* If \( H \neq \emptyset \), we will apply \( \text{FLUSH-H} \). Since we know the oldest element of \( H \) is \( v \), we can treat \( H \) as \( v \cdot H' \). First, this means \( B_\rho = \langle l, S, H' \rangle \). Second, \( B^2 = v \cdot H' \cdot X \) and \( B_\sigma^2 = H' \cdot X \). If \( X = \emptyset \), then \( B_\sigma(x) = \langle l, \emptyset, H' \rangle \). Since \( X \in ESeq(S) \), this also implies that \( S = \emptyset \), so \( B_\rho(x) = \langle l, \emptyset, H' \rangle = B_\sigma(x) \) and we are done. Otherwise, \( X = w \cdot X' \) and \( B_\sigma^2 = H' \cdot w \cdot X' \) for some \( w \). Since \( X \in ESeq(S) \), there are now, again, two options. If \( w \) appears in \( X' \) then \( X' \in ESeq(S) \), otherwise \( X' \in ESeq(S) \).
\( \{w\} \). The first case implies \( B_\sigma(x) \sqsubseteq_b (l, S \setminus \{w\}, H' \cdot w) \), and the second case implies \( B_\sigma(x) \sqsubseteq_b (l, S, H') \). In both cases, as \( w \in S \), we can move it into the set, and get that \( B_\sigma(x) \sqsubseteq_b (l, S, H') = B_\rho(x) \) as required.

* If \( H = \epsilon \) and there are at least two instances of \( v \) in \( B^3 \), we will apply FLUSH-SN. Since \( v \in B^3 \) and \( H = \epsilon \), necessarily \( v \in S \), so FLUSH-SN is enabled. From the abstract semantics, \( B_\rho(x) = (l, S, \epsilon) \). Since \( H = \epsilon \), \( B^3 = X = v \cdot X' \). So \( B^3_\sigma(x) = X' \). Since \( B^3 \) contains \( v \) at least twice, \( X' \) contains \( v \) at least once. So \( X' \) contains instances of exactly the same elements as \( B^3 \), and thus \( X' \in E\text{Seq}(S) \). This means that \( B_\sigma(x) \sqsubseteq_b (l, S, \epsilon) = B_\rho(x) \).

* If \( H = \epsilon \) and there is exactly one instance of \( v \) in \( B^3 \), we will apply FLUSH-SD. Again, necessarily \( v \in S \). To show that FLUSH-SD is enabled, we need to prove that \( \neg(v = l \land S \neq \{v\}) \). Assume that \( v = l \). This means that either \( B^3 = v \) or \( B^3 = v \cdot W \cdot v \) for some sequence \( W \). Since \( v \) appears only once in \( B^3 \), only the first option is possible. This implies that \( B = (v, \{v\}, \epsilon) \), so \( S = \{v\} \) as required.

Now, armed with this lemma, we can prove that the overall semantics are sound.

**Theorem 5.4.3** Let \( \Sigma^3 \in C \) be a set of concrete states. Let \( \tau^3 \) be the concrete PSO transformer, and \( \tau \) the abstract partial-coherence transformer for some bound \( k \). Then \( \alpha(\tau(\Sigma^3)) \sqsubseteq \tau(\alpha(\Sigma^3)) \).

**Proof:** Let \( \sigma \in \alpha(\tau(\Sigma^3)) \) be an abstract state. By definition of \( \sqsubseteq \), we need to show that there exists some \( \rho \in \tau(\alpha(\Sigma^3)) \) such that \( \sigma \sqsubseteq_\alpha \rho \).

By definition of \( \alpha \), \( \sigma \in \bigsqcup \{\hat{\alpha}(\sigma^2) \mid \sigma^2 \in \tau^3(\Sigma^3)\} \), and by definition of \( \sqsubseteq_\alpha \), for some \( \sigma^2 \in \tau^3(\Sigma^3) \), \( \sigma = \hat{\alpha}(\sigma^2) \). Since \( \tau^3 \) is defined as a union, this means
there is some \( \omega^2 \in \Sigma^2 \) s.t. \( \sigma^2 \in \tau(\omega^2) \), that is the transition from \( \omega^2 \) to \( \sigma^2 \) is legal according to the concrete semantics. Let \( \omega = \hat{\alpha}(\omega^2) \). Then there exists some \( \omega' \) s.t. \( \omega \subseteq_s \omega' \) and \( \omega' \in \alpha(\Sigma^2) \) (possibly \( \omega' = \omega \)).

By Lemma 5.4.2 there exists an abstract state \( \rho \) s.t. the transition from \( \omega' \) to \( \rho \) is a legal abstract transition \( (\rho \in \tau(\omega')) \), and \( \sigma \subseteq_s \rho \). If \( \rho \in \tau(\alpha(\Sigma^2)) \) then we are done. If not, then this can only happen because due to the antichain construction there is some \( \rho' \) s.t. \( \rho \subseteq_s \rho' \) and \( \rho' \in \tau(\alpha(\Sigma^2)) \). By transitivity, \( \sigma \subseteq_s \rho' \) and again, we are done.

The abstraction presented above may seem needlessly complicated. One possible objection is that naively abstracting the ordered buffer to an unordered set would have sufficed for verification and inference. We empirically show this is not the case in Chapter 6. Here, we would like to give some intuition for why the naive set abstraction fails.

Consider a version of Peterson’s algorithm with fences shown in Fig. 5.9. Under standard concrete semantics of PSO, those fences guarantee that it is impossible for both processes to be concurrently executing Line 13. Let us examine a possible execution of this algorithm under an abstract memory model where order and recency are not maintained, and the buffer \( B_p(v) \) is abstracted to a set \( S_p(v) \).

1. Initially both processes start with empty buffers, and \( ent0 = ent1 = turn = 0 \). 

Figure 5.9: Peterson’s Algorithm with fences that guarantee mutual exclusion under the PSO memory model
2. Process 0 runs through one iteration of the outer loop (executes lines 1-14 inclusively), without performing a flush after line 14.
3. Process 0 then tries to enter the critical section again and executes lines 1-3 inclusively. At this stage, \( S_{p0}(ent0) = \{true, false\} \).
4. Two flush actions are performed on \( S_{p0}(ent0) \), first flushing \( true \) and then \( false \). At this point \( G(ent0) = false \).
5. Process 0 completes entering the critical section.
6. Process 1 loads \( ent0 \) from global store and since \( ent0 \) is \( false \) process 1 also enters the critical section, and the mutual exclusion specification is violated.

The above example would not have been possible had we kept either: i) ordering information via \( H_p (ent0) \) for at least two values (i.e., \( k = 2 \)) or ii) recency information via \( l_p (ent0) \). In the first case, the order in which \( \{true, false\} \) are flushed would have been consistent with the order in which the values were written: we would have first flushed \( false \) and then \( true \). In the second case, the fence in line 4 would have forced fully flushing \( S_{p0}(ent0) \), resulting in writing out the most recent value (i.e., \( G(ent0) = true \)). While for this program we could have used either \( l_p (ent0) \) or \( H_p (ent0) \) with \( k = 2 \), in other examples both of these refinements with respect to a simple set are required.

**May-Contain Abstraction**

In the abstraction presented above two abstract buffers \( B_1 = \langle l, S_1, H \rangle \) and \( B_2 = \langle l, S_2, H \rangle \), with the same \( H \) element are incomparable if they differ on the contents of their unordered sets \( (S_1 \neq S_2) \). This avoids values from appearing in the set “out of thin air“, as we can never introduce new values by overapproximating int. However, it sometimes leads to distinctions between abstract states that are more precise than necessary. We observe that a more efficient abstraction can be obtained without a significant sacrifice in precision by merging such states. In Chapter 6 we show that combining such states leads to a more scalable abstraction while keeping a sufficient level of precision.

The one distinction we always wish to preserve regarding the \( S \) component is the difference between an empty and a non-empty buffer. This is because many of our semantic rules distinguish between those two cases, and
losing this distinction may lead to a serious deterioration in precision (for example, unnecessarily enabling both the LOAD-B and LOAD-G rules). To capture this, we change the definition of $\sqsubseteq_b$ as follows:

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

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

1. $(S_1 \neq \emptyset \lor H_1 \neq \varepsilon) \land (\exists t. H_2 = head(H_1, t) \land S_2 \supseteq S_1 \cup UTail(H_1, t))$

2. $(H_1 = H_2 = \varepsilon) \land (S_1 = S_2 = \emptyset)$

The first condition is similar to the definition of $\sqsubseteq_b$, but only requires that $S_1 \cup UTail(H_1, t)$ to be a subset of $S_2$, as opposed to equality. As the first condition explicitly excludes the empty buffer, the second is required to handle the case of two empty buffers, so as to preserve reflexivity.

**Lemma 5.4.4** $\sqsubseteq_b \circ$ is a partial order.

**Proof:**

- **Reflexivity:** Let $B = \langle l, S, H \rangle$. If $H = \varepsilon$ and $S = \emptyset$, then condition 2 holds, and $B \sqsubseteq_b B$. Otherwise, taking $t = |H|$, we have $head(H, t) = H$ and $S \cup UTail(H, t) = S \cup \emptyset = S$.

- **Antisymmetry:** Let $B_1 = \langle l_1, S_1, H_1 \rangle$, $B_2 = \langle l_2, S_2, H_2 \rangle$ s.t. $B_1 \sqsubseteq_b B_2$ and $B_2 \sqsubseteq_b B_1$. First, $l_1 = l_2$. If $H_1 = H_2 = \varepsilon$ and $S_1 = S_2 = \emptyset$, then we are done. Also, there exist $t_1, t_2$ s.t. $H_1 = head(H_2, t_1)$ and $H_2 = head(H_1, t_2)$. This means $|H_1| = |H_2|$, and $t_1 = t_2 = |H_1|$, so $H_1 = H_2$. Similarly, $S_2 \subseteq S_1$, so $S_1 = S_2$.

- **Transitivity:** Let $B_1 = \langle l_1, S_1, H_1 \rangle$, $B_2 = \langle l_2, S_2, H_2 \rangle$, $B_3 = \langle l_3, S_3, H_3 \rangle$ s.t. $B_1 \sqsubseteq_b B_2$ and $B_2 \sqsubseteq_b B_3$. First, $l_1 = l_2$ and $l_2 = l_3$ so $l_1 = l_3$. Now, if for one of the pairs condition (1) holds, then those two buffers are equal, and we are done. So, assume condition (2) holds for both pairs. So there exist $t_1, t_2$ s.t. $H_2 = head(H_1, t_1)$, $H_3 = head(H_2, t_2)$, and by definition of head, we have $H_3 = head(H_1, t_1 + t_2)$. Also, $S_1 \cup UTail(H_1, t_1) \subseteq S_2$ and $S_2 \cup UTail(H_2, t_2) \subseteq S_3$. Similarly to the case with head, $UTail(H_1, t_1) \cup UTail(H_2, t_2) = UTail(H_1, t_1 + t_2)$. This means $S_1 \cup UTail(H_1, t_1 + t_2) \subseteq S_3$, so condition (2) holds and $B_1 \sqsubseteq_b B_3$ as required.
The order $\sqsubseteq_s$ is then defined exactly as before, but with respect to $\sqsubseteq_b$ instead of $\sqsubseteq_b$. The abstract domain $A_s$ is then, again, the set of antichains of elements of $A$, but with respect to $\sqsubseteq_s$. This seemingly small formal change has several important effects on both the abstraction function and on the technical properties of $A_s$.

The abstraction function is defined exactly as before. However, the change of domain forces the intuitive meaning of the abstraction, and the definition of the concretization function to change. To see this, consider the abstract buffers $B_1 = \langle l, S, \epsilon \rangle$, $B_2 = \langle l, S \cup \{ v \}, \epsilon \rangle$ where $S \neq \emptyset$. Under the previous abstraction, $B_1 \sqsubseteq_b B_2$. On the other, we need to preserve the property that $\hat{\gamma}_B$ is monotonic: if $B_1 \sqsubseteq_b B_2$, then $\hat{\gamma}_B(B_1) \subseteq \hat{\gamma}_B(B_2)$. It is easy to see that this can be achieved by removing the “at least once” requirement from the definition of $\hat{\gamma}_B$. That is, $\hat{\gamma}_B(B_1)$ is simply the set of all non-empty buffers ending in $l$ that can be constructed from $S$. The extension of this to buffers with a non-empty $H$ element is trivial, and we get that

$$\hat{\gamma}_B^\diamond(\langle l, S, H \rangle) = \{ H \cdot X \mid X \in Seq(S) \land last(H \cdot X) = l \}$$

Due to this property, we refer to the previous abstraction as the must-contain abstraction ($\Box C$) and to the abstraction under discussion as may-contain abstraction ($\diamond C$).

From the technical point of view, the join operator $\sqcup^\diamond$ on $A$ is quite different from the previous version: while the formal definition is the same, the behavior (which depends on the partial order $\sqsubseteq_s$) is not. In the previous case, $\sqsubseteq_s$ was a union of linear orders. This means we always had $(\Sigma_1 \sqcup \Sigma_2) \subseteq (\Sigma_1 \sqcup \Sigma_2)$, with the only difference between the join and the union being the pruning of non-maximal elements. This is no longer true for $A_s$, as the join may merge two non-comparable (by $\sqsubseteq_s$) abstract states into their least upper bound when such a bound exists. For example, consider three states $\sigma_1, \sigma_2, \sigma_3$ that are identical with the exception of some buffer $B(p)$. For the pair $p, x$ the three buffers are $B_1 = \langle a, \{ a, b \}, \epsilon \rangle$, $B_2 = \langle a, \{ a, c \}, \epsilon \rangle$ and $B_3 = \langle a, \{ a, b, c \}, \epsilon \rangle$. Clearly, $\sigma_1$ and $\sigma_2$ are non-comparable by $\sqsubseteq_s$, but
\[ H(x) = \epsilon \quad v \in S(x) \quad (\text{FLUSH-NE}) \]
\[ G'(x) = v \]
\[ H(x) = \epsilon \quad S(x) \neq \emptyset \quad (\text{FLUSH-E}) \]
\[ G'(x) = l(x) \quad S'(x) = \emptyset \quad l'(x) = \perp \]

Figure 5.10: May-contain flush semantics

\( \{\sigma_1\} \uplus^\circ \{\sigma_2\} = \{\sigma_3\} \not\subseteq \{\sigma_1, \sigma_2\} \).

Finally, the new abstraction also implies a change to the abstract transformer. In the \( \square C \) abstraction, flushes from the set \( S \) were split into two cases: FLUSH-SD to represent flushing the last instance of some value \( v \) from the buffer, and FLUSH-SN to represent an instance that is not the last one. The case split for the \( \circ C \) abstraction needs to be slightly different. The new flush semantics are shown in Fig. 5.10. The rule FLUSH-NE covers the case in which after a flush \( S \) remains non-empty, while FLUSH-E represents flushing the only remaining element of the (concrete) buffer. As before, it’s possible for both types of flush rules to be enabled for the same buffer. Note that the rule FLUSH-H remains unchanged.

To show the soundness of the \( \circ C \) transformer, we need to prove the analogue of Theorem 5.4.3. The proof is very similar to the original, except some slight technical changes, and the fact it requires the analogue of Lemma 5.4.2 for the \( PJ \) transformer. In the lemma below, we will omit the \( \circ \) marks, as it refers only to the \( \circ C \) abstraction.

**Lemma 5.4.5** Let \( \omega^\circ, \sigma^\circ \) be concrete states such that \( \sigma^\circ \in \tau^\circ(\omega^\circ) \). Let \( \sigma = \hat{\alpha}(\sigma^\circ) \). Let \( \omega = \hat{\alpha}(\omega^\circ) \), and \( \nu \) some abstract state s.t. \( \omega \sqsubseteq_s \nu \). Then there exists \( \rho \in \tau(\nu) \) s.t. \( \sigma \sqsubseteq_s \rho \).

**Proof:** To fully prove this lemma, we need to show that we can find an abstract transition that over-approximates any concrete transition. We will, however, skip the proofs for all rules but FLUSH. While the fact all other inference rules are the same as in the \( \square C \) semantics does not mean the proofs are identical, they are, in fact, quite similar. In this proof, we use the same notation as in Lemma 5.4.2.

Assume the rule FLUSH was applied to \( \omega^\circ \). As before, we decide on which abstract rule to invoke according to the contents of \( B^\circ \) and \( B_\nu(x) = (l, S, H) \)
• If $H \neq \epsilon$, we will apply FLUSH-H. Since we know the last element of $H$ is $v$, we can again treat $H$ as $v \cdot H'$. This means $B_\rho = \langle l, S, H' \rangle$, $B^2 = v \cdot H' \cdot X$ and $B^2_\sigma = H' \cdot X$, where $X \in Seq(S)$. If $H' \neq \epsilon \land S \neq \emptyset$, then this implies directly that $B_\sigma \sqsubseteq_b \langle l, S, H' \rangle = B_\rho$. Otherwise, necessarily $X = \epsilon$ and $B_\sigma = B_\rho = \langle l, \emptyset, \epsilon \rangle$, as desired.

• If $H = \epsilon$ and $B^2 \neq v$, we will apply FLUSH-NE. First, $B_\rho = B_\nu = \langle l, S, H \rangle$. Second, $B^2 = v \cdot X$ where $X \in Seq(S)$ and $X \neq \epsilon$. So $B^2_\sigma = X \neq \epsilon$, and immediately $B_\sigma \sqsubseteq_b \langle l, S, H \rangle$.

• If $H = \epsilon$ and $B^2 = v$, we will apply FLUSH-E. First, $B_\rho = \langle \bot, \emptyset, \epsilon \rangle$. Second, $B^2_\sigma = \epsilon$, so $B_\sigma = \langle \bot, \emptyset, \epsilon \rangle = B_\rho$, and this concludes the proof.

\[ \square \]

5.5 Abstract Fence Inference

Our next goal is to apply Algorithm 1 to abstract state-spaces. We wish to prove the that the following holds:

Claim 5.5.1 Let $P$ be a program and $\langle \sigma_0^\natural, \Sigma_P^\natural, T_P^\natural \rangle$, $\langle \sigma_0, \Sigma_P, T_P \rangle$ the concrete and abstract transition systems for $P$. Let $\psi = \bigvee \{ av(\sigma) \mid \sigma \in \Sigma_P \land \sigma \not\models S \}$, and $\delta \subseteq V$. If $\delta \models \psi$ then $\forall \sigma^\natural \in \Sigma_P^\natural. (\sigma^\natural \not\models S \implies \delta \not\models \sigma^\natural)$

The premise of the above claim needs to hold in the abstract transition system, while the conclusion holds in the concrete one. Informally, this claim means that if we run the inference algorithm on the abstract transition system instead of the concrete one, enforcing the resulting constraints in the (by definition, “concrete”) program is still safe.

While this may not be obvious at first glance, the claim above, as stated, is not in fact well-defined. That is because it involves $av(\sigma)$, which is defined in terms of $\chi(t)$. However, $\chi(t)$ itself depends on the memory model, and we have not defined it for either the must- or the may-contain abstractions. Before defining it for the specific models, we can give a simple property of $\chi$, s.t. that if it holds, the entire claim holds as well.

Let $\chi^\natural: T_P^\natural \rightarrow 2^V$ be the prevent function in the concrete transition system, and $\chi: T_P \rightarrow 2^V$ the prevent function in the abstract transition system. Let $t = \sigma_1 \rightarrow \sigma_2$ be a transition between two abstract states in
Finally, let $\Gamma(t)$ be a set of concrete transitions, such that $t^\natural = \sigma_1^\natural \rightarrow \sigma_2^\natural$ is in $\Gamma(t)$ if all of the following hold:

- $\sigma_1^\natural \in \gamma(\sigma_1)$
- $\sigma_2^\natural \in \gamma(\sigma_2)$
- $\sigma_2^\natural \in \tau^\natural(\sigma_1^\natural)$

Note that $t^\natural \in \Gamma(t)$ is not required to appear in $T_P$. Informally, $\Gamma(t)$ lifts $\gamma$ to transitions: it is the set of all possible concrete transitions represented by the abstract transition $t$.

We say $\chi$ is precise with respect to $\chi^\natural$ if for any $t \in T_P$, and for any $t^\natural \in \Gamma(t)$, $\chi(t) \implies \chi^\natural(t^\natural)$.

**Theorem 5.5.2** If $\chi$ is precise w.r.t $\chi^\natural$ then Claim 5.5.1 holds.

**Proof:** Suppose by contradiction there is some $\sigma^\natural \in \Sigma_P$ s.t. $\delta \not\vdash \sigma^\natural$. Then there is some trace $\pi^\natural = t_0^\natural, \ldots, t_k^\natural$ where $t_i^\natural = \sigma_i^\natural \rightarrow \sigma_{i+1}^\natural$, $\sigma_{k+1}^\natural = \sigma^\natural$ in $T_P$ s.t. $\delta \not\vdash \chi^\natural(\pi^\natural)$. Since the abstraction is sound, we know there is a corresponding trace $\pi$ in $T_P$, where $\sigma_i^\natural \in \gamma(\sigma_i)$. By Theorem 4.1.6, we know that $\delta \not\vdash \sigma$, so $\delta \models \chi(\pi)$. This means that for any $i \leq k$, $\delta \models \chi(t_i)$. By construction, for any $i$, $\sigma_i^\natural \in \gamma(\sigma_i)$, and because $\chi$ is precise, we have $\chi(t_i) \implies \chi^\natural(t_i^\natural)$. So for any $i$, $\delta \models \chi^\natural(t_i^\natural)$, and hence $\delta \models \chi^\natural(\pi^\natural)$, a contradiction.

The $\chi$ function for the must-contain abstraction can be defined as follows:

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

$$\chi(t) = \bigvee \{ [l < l_i] \mid \exists x, v. (l, v) \in \text{Set}(Q_{\sigma(t)}(p)(x))\}$$

This definition of $\chi(t)$ is a natural adaptation of the definition of $\chi^\natural(t)$ given earlier in this chapter. To see this, consider a concrete buffer $B^\natural$ and its abstraction $B = \langle l, S, H \rangle = \hat{\alpha}_B(B^\natural)$. Then $v \in \text{Set}(B^\natural)$ if and only if $v \in S \cup \text{Set}(H)$. This fact is used to prove the following claim:

**Claim 5.5.3** The $\chi$ function defined above is precise w.r.t the $\chi^\natural$ function of concrete PSO.

**Proof:** For this proof, it is again more convenient to treat $\chi$ and $\chi^\natural$ as sets of disjuncts, instead of as formulae. In this case, the precision property translates to $\chi(t) \subseteq \chi^\natural(t^\natural)$ for all appropriate $t$ and $t^\natural$. Let $t = \sigma_1 \rightarrow \sigma_2$
be a transition between two abstract states, and \( p = \text{proc}(t) \). Let \( \sigma_1^1 \in \gamma(\{\sigma_1\}), \sigma_2^1 \in \gamma(\{\sigma_2\}) \), s.t. \( \sigma_2^1 \in \tau^2(\{\sigma_1^1\}) \). Let \( t^2 = \sigma_1^1 \rightarrow \sigma_2^1 \). Finally, let \( w = [l_w < l_i] \in \chi(t) \). Because \( w \in \chi(t) \), we know that there exist some \( x, v \) s.t. \( \langle l_w, v \rangle \in Q_{\sigma_1}(p)(x) \). Let \( B = \langle l, S, H \rangle = B_{\sigma_1}(p)(x) \) be the appropriate abstract buffer of \( \sigma_1 \). We know that \( \langle l_w, v \rangle \) appears in either \( S \) or \( H \). Thus, by definition of \( \tilde{\gamma}_B \), for any \( B^3 \in \tilde{\gamma}_B(B), \langle l_w, v \rangle \) appears in \( B^3 \). Since \( \sigma_2^1 \in \gamma(\{\sigma_1\}) \), \( B_{\sigma_1}^2(p)(x) \in \tilde{\gamma}_B(B) \), so \( \langle l_w, v \rangle \in \text{Set}(B_{\sigma_1}^2(p)(x)) \). So, \( w \in \chi^2(t) \). Since this holds for any \( w \in \chi(t) \), we have that \( \chi(t) \subseteq \chi^2(t) \) as required.

Unfortunately, for the may-contain abstraction, the above \( \chi(t) \) is no longer precise. Consider the abstract buffer \( B^o = \langle a, \{a, b\}, \epsilon \rangle \), and some concrete buffer \( B^3 \in \tilde{\gamma}_B(B^o) \). Assume the value \( a \) was written by a store at label \( l_a \), and the value \( b \) by a store at label \( l_b \). Due to the nature of the \( \Theta_C \) abstraction, \( b \) is not guaranteed to appear in \( B^3 \). So, we can take \( B^3 = "a" \). Now, let \( t \) be a transition by process \( p \) from a state \( \sigma \) s.t. \( B(p)(x) = B^o \), and we use \( \chi(t) = [l_a < l_i] \lor [l_b < l_i] \) as before. This constraint would be satisfied by enforcing \( [l_b < l_i] \). Unfortunately, enforcing \( [l_b < l_i] \) would not avoid a concrete transition from a state with the buffer \( B^3(p)(x) = B^3 \), as it does not contain any values written by the statement at \( l_b \).

To overcome this issue, we define \( \chi^o(t) \) in a more conservative manner:

\[
\chi^o(t) = \bigwedge \left\{ [l < l_i] \mid \exists x, v. \langle l, v \rangle \in \text{Set}(Q_{\text{src}(t)}(p)(x)) \right\}
\]

It is easy to show that this is precise w.r.t \( \chi(t)^o \).

**Claim 5.5.4** \( \chi^o \) is precise w.r.t the \( \chi \) function of concrete PSO.

**Proof:** As before, let \( t = \sigma_1 \rightarrow \sigma_2 \) be a transition between two abstract states, and \( p = \text{proc}(t) \). Let \( \sigma_1^1 \in \gamma(\{\sigma_1\}), \sigma_2^1 \in \gamma(\{\sigma_2\}) \), s.t. \( \sigma_2^1 \in \tau^2(\{\sigma_1^1\}) \). Let \( t^2 = \sigma_1^1 \rightarrow \sigma_2^1 \). If \( \chi^2(t^2) = \text{ff} \), then this holds vacuously. So, assume it is not. Let \( \delta \) be an assignment s.t. \( \delta \models \chi^o(t) \). Since \( \chi^o(t) \) is a conjunction, any such assignment \( \delta \) must assign \( \text{tt} \) to all variables \( w = [l < l_i] \) s.t. for some \( v, x, \langle l, v \rangle \in Q_{\sigma_1}(p)(x) \). Now, consider \( B_{\sigma_1}^2(p)(x) \). If it is empty then \( \chi^2(t^2) = \text{ff} \), a contradiction. So, assume it is not. The only elements it may contain are those in \( Q_{\sigma_1}(p)(x) \), which implies at least one \( w \) with the properties given above appears as a disjunct in \( \chi^2(t^2) \), which means \( \delta \models \chi^o(t^2) \), as required. \( \square \)
Chapter 6

Experimental Results

Beware of bugs in the above code; I have only proved it correct, not tried it. — Donald Knuth

We have implemented our algorithm in a pair of tools called FENDER and BLENDER. FENDER is a direct implementation of the fence inference algorithm of Chapter 4 for the RLX framework. In BLENDER we adapted the implementation to work with abstract memory models from Chapter 5. Both FENDER and BLENDER were implemented in Java, and use external libraries to find satisfying assignments to constraint formulae. FENDER represents the formulae explicitly and solves them using the Sat4J library. BLENDER uses the JavaBDD library to represent the formulae as BDDs.

In this chapter we describe two sets of experiments, one using FENDER and the other using BLENDER. The goals of the two sets of experiments were somewhat different.

1. The goal of the first set (using FENDER) was to establish the practical feasibility of the inference algorithm. This set of experiments examines the inference results for realistic concurrent data structures and the interaction between the algorithm and the underlying memory model.

2. The goal of the second set (using BLENDER) was to examine the behavior and properties of real concurrency primitives under abstract memory models. Here the focus is not on the inference algorithm but on the trade-offs between the abstract models and their parameters in the context of fence inference.
6.1 Fence Inference in the RLX framework

Methodology

This set of experiments was performed on an IBM xSeries 336 server with 4 Intel Xeon 3.8GHz processors and 5GB RAM, running a 64-bit version of Red Hat Enterprise Linux. In the experiments, we varied the following parameters:

(i) Input Algorithms - 5 concurrent data structures and one mutual exclusion algorithm.
(ii) Client Program - for each input, we used clients of varying size and complexity.
(iii) Memory Model - 3 relaxed models and the sequentially consistent model as a baseline.
(iv) Specification - for several benchmarks, there is more than one reasonable specification.
(v) A bound on the execution buffer, when required.

Algorithms We applied fender to various challenging state-of-the-art concurrent algorithms:

- **Treiber**: Treiber’s lock-free stack [75].
- **MSN**: Michael & Scott’s lock-free queue [57].
- **Chase-Lev WSQ**: Chase & Lev’s work-stealing queue [20].
- **LIFO WSQ**: LIFO idempotent work-stealing queue [58].
- **VYSet**: Vechev & Yahav’s concurrent list-based set [76].
- **Dekker**: Dekker’s mutual exclusion [25].

Clients As explained in Chapter 4, our algorithm requires the input program to be finite state. Therefore, if we wish to apply the algorithm to data structures, the input must be the data structure implementation in conjunction with a client. For each of the above data structures, we ran fender with several clients. A client typically consists of 2 or 3 processes, with each process invoking several data structure operations. Below, we use the term “program” to refer to the combination of a data structure and a client. In most cases, the clients were chosen manually (sometimes duplicating the choices made in [15]). However, fender permits exhaustive exploration of bounded clients that consist of a (bounded) sequence of initialization operations followed by (bounded) sequences of operations performed in parallel.
**Memory Models** We ran our experiments under RLX implementations of TSO, PSO and speculation-free RMO. We also ran the same experiments for SC, to serve as a baseline for the state-space sizes.

**Specification** In our experiments we always consider safety specifications, realized as state invariants on the program’s final state. Ideally, we would like to use linearizability as the specification. However, linearizability under relaxed memory models is not well-defined (See [13] for a short explanation). Since producing specifications was not the focus of our work, we manually examined the behavior of our inputs, and wrote state invariants that correspond to sequentially consistent behavior of the data structures.

**Bound on the Execution Buffer** Even given a client, two of our benchmarks (VYSet and Dekker) are not finite-state under RMO, due to potentially infinite execution buffers. Therefore we were forced to add an arbitrary bound on the size that the execution buffer is allowed to reach. Of course, arbitrary bounding means we could not perform a sound verification (or inference) procedure, but we believe the results for those two benchmarks to be interesting nevertheless.
<table>
<thead>
<tr>
<th>Initial State</th>
<th>Client</th>
<th></th>
<th>Time (sec.)</th>
<th>States</th>
<th>Transitions</th>
<th>#C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSN empty</td>
<td>e</td>
<td>d</td>
<td>∞</td>
<td>0.83</td>
<td>1219</td>
<td>2071</td>
</tr>
<tr>
<td>MSN empty</td>
<td>e</td>
<td>e</td>
<td>∞</td>
<td>1.78</td>
<td>4934</td>
<td>12670</td>
</tr>
<tr>
<td>MSN empty</td>
<td>ee</td>
<td>dd</td>
<td>∞</td>
<td>5.21</td>
<td>24194</td>
<td>61514</td>
</tr>
<tr>
<td>MSN empty</td>
<td>ed</td>
<td>ed</td>
<td>∞</td>
<td>13.05</td>
<td>86574</td>
<td>242822</td>
</tr>
<tr>
<td>MSN empty</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>∞</td>
<td>9.26</td>
<td>59119</td>
</tr>
<tr>
<td>MSN empty</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>∞</td>
<td>31.43</td>
<td>233414</td>
</tr>
<tr>
<td>ChaseLev empty</td>
<td>pppt</td>
<td>ttt</td>
<td>ss</td>
<td>∞</td>
<td>97.22</td>
<td>386283</td>
</tr>
<tr>
<td>WSQ empty</td>
<td>tt</td>
<td>ttt</td>
<td>pt</td>
<td>tp</td>
<td>ss</td>
<td>∞</td>
</tr>
<tr>
<td>WSQ empty</td>
<td>pppt</td>
<td>tt</td>
<td>ttt</td>
<td>pp</td>
<td>ss</td>
<td>∞</td>
</tr>
<tr>
<td>WSQ empty</td>
<td>tt</td>
<td>tttt</td>
<td>tt</td>
<td>pp</td>
<td>ss</td>
<td>∞</td>
</tr>
<tr>
<td>WSQ empty</td>
<td>tt</td>
<td>tttt</td>
<td>ttt</td>
<td>pp</td>
<td>ss</td>
<td>∞</td>
</tr>
<tr>
<td>&quot;LIFO&quot; 2/2</td>
<td>tp</td>
<td>ss</td>
<td>∞</td>
<td>0.69</td>
<td>2151</td>
<td>3190</td>
</tr>
<tr>
<td>WSQ 2/2</td>
<td>tpt</td>
<td>ss</td>
<td>∞</td>
<td>1.94</td>
<td>9721</td>
<td>16668</td>
</tr>
<tr>
<td>WSQ 2/2</td>
<td>ptp</td>
<td>ss</td>
<td>∞</td>
<td>11.41</td>
<td>89884</td>
<td>195246</td>
</tr>
<tr>
<td>WSQ 2/2</td>
<td>ptt</td>
<td>ss</td>
<td>∞</td>
<td>11.31</td>
<td>85104</td>
<td>198353</td>
</tr>
<tr>
<td>WSQ 1/1</td>
<td>ptt</td>
<td>ss</td>
<td>∞</td>
<td>4.07</td>
<td>23913</td>
<td>48997</td>
</tr>
<tr>
<td>Dekker -</td>
<td>-</td>
<td>1</td>
<td>0.64</td>
<td>1388</td>
<td>2702</td>
<td>2</td>
</tr>
<tr>
<td>Dekker -</td>
<td>-</td>
<td>10</td>
<td>2.13</td>
<td>7504</td>
<td>14477</td>
<td>2</td>
</tr>
<tr>
<td>Dekker -</td>
<td>-</td>
<td>20</td>
<td>2.71</td>
<td>13879</td>
<td>26422</td>
<td>2</td>
</tr>
<tr>
<td>Dekker -</td>
<td>-</td>
<td>50</td>
<td>5.99</td>
<td>33004</td>
<td>62257</td>
<td>2</td>
</tr>
<tr>
<td>Treiber empty</td>
<td>p</td>
<td>t</td>
<td>∞</td>
<td>1</td>
<td>71</td>
<td>93</td>
</tr>
<tr>
<td>Treiber empty</td>
<td>pt</td>
<td>tp</td>
<td>∞</td>
<td>1.02</td>
<td>3054</td>
<td>6190</td>
</tr>
<tr>
<td>Treiber empty</td>
<td>pp</td>
<td>tt</td>
<td>∞</td>
<td>0.6</td>
<td>1276</td>
<td>2250</td>
</tr>
<tr>
<td>VYSet empty</td>
<td>ar</td>
<td>ra</td>
<td>10</td>
<td>1.98</td>
<td>4079</td>
<td>6247</td>
</tr>
<tr>
<td>VYSet empty</td>
<td>aa</td>
<td>rr</td>
<td>10</td>
<td>4.56</td>
<td>20034</td>
<td>31023</td>
</tr>
<tr>
<td>VYSet empty</td>
<td>ar</td>
<td>ar</td>
<td>10</td>
<td>2.19</td>
<td>6093</td>
<td>9005</td>
</tr>
<tr>
<td>VYSet empty</td>
<td>aa</td>
<td>rr</td>
<td>10</td>
<td>7.98</td>
<td>41520</td>
<td>66533</td>
</tr>
</tbody>
</table>

Table 6.1: Experimental results for the RMO model
| Initial | Client | $|E|$ Bound | RMO States | RMO Edges | PSO States | PSO Edges | TSO States | TSO Edges | SC States | SC Edges |
|---------|--------|-----------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| MSN     | empty  | e|d | $\infty$ | 1219 | 2671 | 2 | 455 | 743 | 1 | 228 | 316 | 0 | 146 | 180 |
|         | empty  | e|e | $\infty$ | 4934 | 12670 | 1 | 2678 | 6354 | 1 | 586 | 994 | 0 | 252 | 328 |
|         | empty  | e|e|d | $\infty$ | 24194 | 61514 | 3 | 7025 | 13689 | 2 | 1724 | 2512 | 0 | 1029 | 1325 |
|         | empty  | e|d|d | $\infty$ | 86574 | 242822 | 2 | 15450 | 35362 | 2 | 2476 | 3972 | 0 | 1538 | 2126 |
|         | empty  | e|d|e | $\infty$ | 59119 | 167067 | 4 | 11023 | 24362 | 2 | 2570 | 4010 | 0 | 1541 | 2073 |
|         | empty  | e|d|d | $\infty$ | 233414 | 653094 | 3 | 51990 | 119050 | 2 | 9638 | 16822 | 0 | 4928 | 7632 |
| Chase-Lev WSQ | empty  | pppt(t|pt|ss) | $\infty$ | 386283 | 1030857 | * | 74533 | 256613 | * | 12348 | 20004 | * | 4961 | 6740 |
|         | empty  | tttt(pt|t|ss) | $\infty$ | 1048498 | 2819355 | * | 124455 | 255390 | * | 6418 | 9380 | * | 3101 | 4069 |
|         | empty  | pppt(t|tt|p|ss) | $\infty$ | 281314 | 878880 | * | 66960 | 241814 | * | 10564 | 16317 | * | 4199 | 5700 |
|         | empty  | tttt(t|pp|t|ss) | $\infty$ | 1325858 | 4150650 | * | 361855 | 1080835 | * | 9878 | 13956 | * | 3473 | 4537 |
|         | empty  | tttt(t|pp|t|pp) | $\infty$ | 280396 | 698398 | * | 29573 | 54969 | * | 9197 | 14499 | * | 4760 | 6455 |
| "LIFO" WSQ | 2/2    | tp|ss | $\infty$ | 2151 | 3190 | 2 | 882 | 1171 | 1 | 676 | 852 | 0 | 570 | 694 |
|         | 2/2    | tpt|ss | $\infty$ | 9721 | 16668 | 2 | 3908 | 5811 | 1 | 2256 | 3116 | 0 | 1410 | 1786 |
|         | 2/2    | ptpt|ss | $\infty$ | 89884 | 195246 | 3 | 31289 | 64133 | 3 | 4045 | 5688 | 0 | 2317 | 3007 |
|         | 1/1    | ptt|ss | $\infty$ | 85104 | 198353 | 4 | 29920 | 62020 | 3 | 4130 | 5987 | 0 | 2198 | 2866 |
| Dekker  | *      | * | 1 | 1388 | 2702 | 2 | 1388 | 2702 | 2 | 489 | 674 | 1 | 388 | 490 |
|         | *      | * | 10 | 7504 | 14477 | 2 | 7504 | 14477 | 2 | 2560 | 3750 | 1 | 388 | 490 |
|         | *      | * | 20 | 13879 | 26422 | 2 | 13879 | 26422 | 2 | 4845 | 7115 | 1 | 388 | 490 |
|         | *      | * | 50 | 33004 | 62257 | 2 | 33004 | 62257 | 2 | 11770 | 17210 | 1 | 388 | 490 |
| Treiber  | empty  | p|t | $\infty$ | 71 | 93 | 2 | 71 | 93 | 2 | 43 | 48 | 0 | 36 | 38 |
|         | empty  | pt|tp | $\infty$ | 3054 | 6190 | 2 | 3041 | 6167 | 2 | 407 | 609 | 0 | 392 | 482 |
|         | empty  | p|tt | $\infty$ | 1276 | 2250 | 2 | 1276 | 2250 | 2 | 325 | 407 | 0 | 270 | 323 |
| VYSet   | empty  | ar|ra | 10 | 4079 | 6247 | 2 | 4079 | 6247 | 2 | 1088 | 1308 | 0 | 1088 | 1308 |
|         | empty  | a|ar | 10 | 20034 | 31623 | 2 | 20034 | 31623 | 2 | 1168 | 1411 | 0 | 1168 | 1411 |
|         | empty  | ar|ar | 10 | 6093 | 9905 | 2 | 6093 | 9905 | 2 | 1671 | 1968 | 0 | 1671 | 1968 |
|         | empty  | aa|rr | 10 | 41520 | 66533 | 2 | 41520 | 66533 | 2 | 3311 | 4072 | 0 | 3311 | 4072 |

Table 6.2: Experimental results for different memory models
Quantitative Results

A summary of our experimental results for RMO is shown in Tab. 6.1. As noted above, several parallel clients were used for each data structure. For each client, the “Initial” and “Client” columns represent the initial state of the data structure and the operations performed by the client respectively. In the “Client” column, “e” represents an enqueue operation, “d” a dequeue, “p” put, “s” steal, “a” add and “r” remove. The “|E|” column represents the bound on the length of execution buffers used for this execution. The “Time” column shows the total analysis time. This includes the state exploration time, the constraint inference time and the time it took to find minimal solutions (SAT solver executing time). Note that in all cases the solving component was negligible. Finally, “#C” is the number of constraints in a minimal solution to the avoid formula for that client. For the Chase-Lev work-stealing queue the constraint formulae were not solved for individual clients. Hence, in that case, individual “#C” values are not given.

In Tab. 6.2 we show a comparison of the performance of Fender for different memory models it supports. On average the number of states for PSO was $\approx 4.5$ times smaller and for TSO $\approx 40$ times smaller than for RMO.

Detailed Results

Below we examine the results for some of our benchmarks in more detail.

**Michael-Scott Queue** The non-blocking queue of [57] is one of few algorithms for which a correct fence placement (for RMO) has been published [15]. We refer to that placement of fences as the “reference placement”. The reference placement uses 7 fences, 4 in enqueue(), and 3 in dequeue(). As [15] notes, all of the fences were found using small test-cases. Our hypothesis was that by running Fender with a small number of test-cases, we can automatically infer the appropriate fences.

Under RMO (which is closest to the model used by [15]), a small set of clients produced 20 different sets of 4 constraints. Using the local fence placement method there are only 4 different ways to implement those sets using 3 fences: 1 fence in enqueue() and 2 in dequeue(). One of those placements was, as expected, a proper subset of the reference placement found in [15], and the others were similar.
void enqueue(queue_t *queue, value_t value)
{
    node_t *node, *tail, *next;
    node = new_node();
    node->value = value;
    node->next = 0;
    fence("store-store");
    while (true) {
        tail = queue->tail;
        fence("load-load");
        next = tail->next;
        fence("load-load");
        if (tail == queue->tail)
            if (next == 0) {
                if (cas(&tail->next,
                        (unsigned) next, (unsigned) node))
                    break;
            } else
                cas(&queue->tail,
                    (unsigned) tail, (unsigned) next);
    }
    fence("store-store");
    cas(&queue->tail,
        (unsigned) tail, (unsigned) node);
}

Figure 6.1: Enqueue operation of the Michael-Scott queue (from [15])
Fig. 6.1 is copied verbatim from [15] and shows the enqueue() method for the algorithm (including 4 of the 7 fences placed using CheckFence). The reference placement contains 7 fences, while our tool inferred only 3 of these 7, which may seem, at first glance, insufficient. However, manual examination of the 4 missing fences confirms that they are in fact redundant in our model.

- The load-load fence on line 10 of Fig. 6.1 prevents the load on line 11 from being executed before the load on line 9 (note that the two loads are data dependent). To the best of our understanding SPARC RMO only allows control speculation, but not data speculation, which means this fence is in fact not necessary.

- The store-store fence on line 22 prevents the CAS on line 23 from being executed before the CAS on line 15. However, this may only happen if the CAS on line 23 is executed speculatively, since its execution is control-dependent on the success of the CAS in line 15. Under RMO operations that write to memory (and, in particular, CAS operations) may not be executed speculatively, so this fence is never needed.

- The two load-load fences on line 12 and on line 57 of the dequeue() code given in [15] enforce the correct execution of a construct meant to solve a certain type of ABA problem that only occurs when immediate reuse of memory is allowed. However, under the assumption of automatic memory management, the statements in lines 13 and 58 are redundant (see [54]). Since the correct execution of these two statements is no longer important, fender correctly omits the two fences that “protect” them.

Under PSO, only sets of two constraints (implementable by a single store-store fence) in the enqueue() method were inferred. This is consistent with the fact that, under this model, loads are not reordered with each other so load-load constraints are unnecessary. Under TSO, no fences were inferred, again consistent with our expectations, and with the claim in [15] that under the x86 memory model (which resembles TSO), no fences should be necessary.

**Idempotent Work-Stealing: LIFO** Michael et al. [58], present several high performance work stealing algorithms that allow a task in the queue to be returned more than once, relaxing the standard queue semantics where a task is returned exactly once. This weakening of the specification allows the
avoidance of expensive store-load fences in the queue owner's operations.
We applied fender to the LIFO version of the idempotent queue.

The implementation description in [58] does not provide a reference fence placement. Rather, it gives constraints on execution, very similar to the constraints we generate. The reference includes 5 such constraints: 3 in the put() operation and 2 in steal().

The data structure specification is more relaxed than sequential consistency, as operations may return results that are not allowed by any sequential execution. Instead of trying to construct a specification that relates the IWSQ to regular work-stealing queues, we explicitly define error states as states in which one of the following occurs:

- Uninitialized access: A shared variable is read before it was ever written.
- Lost tasks: In a final state, the queue is empty but there is some value which was the target of a put() operation but was not returned by any take() or steal() operations.

Under RMO, our tool produced 3 possible sets of 4 maximally permissive constraints, one of which is a proper subset of the constraints as described in the paper [58]. Like in the Michael-Scott Queue, the missing constraint is due to our model disallowing speculation. The reference contains the constraint “Order read in 1 before read in 3” in the steal() operation. However in the code, line 3 is control-dependent on line 2 which, in turn, is data-dependent on line 1. If we remove the control dependency (by manually re-ordering lines 2 and 3 in the code), fender correctly infers this constraint in addition to the rest.

Under PSO, only 3 constraints were inferred, because the constraint which orders a read and the CAS in steal() is no longer required. Under TSO, fender correctly inferred no constraints.

**Classic Work-Stealing: Chase-Lev**

The idempotent work-stealing specification described above allows an element to be returned by several put() and steal() operations. A more “classical” specification of a work-stealing queue allows an element to be returned at most once.

In the case of the Chase-Lev WSQ we used a different technique in choosing clients. This was due to our lower confidence in the results of hand-picked clients, since there exists no published reference placement. Instead, we ran a bounded search procedure that ran our tool with the set of all...
2-process clients that satisfy the following properties:

1. The client has up to 4 “initialization operations”, executed sequentially before the parallel part.
2. The parallel part invokes a total of up to 6 operations.
3. Between 3 and 5 of these operations are invoked by the owner process.

If a particular client’s state space exceeded 2.5 million states, it was terminated and discarded.

We inferred fences for the Chase-Lev WSQ using both the classical and idempotent specifications under the RMO model. The result, for both specifications, was the set of 6 fences shown in Fig. 1.1 in Chapter 1. In Tab. 6.2 we show representative clients that produced useful constraints for the idempotent spec.

For the idempotent specification, under RMO, FENDER inferred a set of 9 constraints which can be implemented using the 6 fences of Fig. 1.1. This is possible because the same fence can enforce several constraints. For example, the fence between lines 9 and 10 in expand() also prevents the reordering of the store on line 10 with the stores on lines 6 and 8. Under PSO, we are left with 6 constraints and 3 fences—all of the fences in steal() are no longer needed. Even under TSO, one fence still remains necessary—it is the store-load fence between lines 4 and 5 in the take() operation. The stronger specification produced the exact same set of constraints.

Dekker As Dekker’s algorithm appears in both of our sets of benchmarks, we defer the discussion to the relevant part of the next section.

6.2 Abstract Memory Models

Methodology

The experiments in this section were conducted on an 8 CPU Xeon 1.6GHz server with 16GB RAM running a 64-bit Sun JVM on Red Hat Linux.

Algorithms The set of benchmarks we use here is quite different from the previous set. The benchmarks were selected based on two main criteria:

1. The benchmarks could not be handled by any previously known approach. For instance, as mutual exclusion algorithms inherently contain benign data races, using techniques like delay set analysis [70]
would result in over-estimation of the required fences. Furthermore, some of the benchmarks — for instance Dek and Fast — contain benign triangular data races (as defined in [60]). Thus, even if we focus squarely on the TSO memory model, we could not use the results of [60] to establish the correctness of the algorithms by focusing only on sequentially consistent executions. Finally, all of the benchmarks contain unbounded spin-loops, and as such, they cannot be handled by bounded methods, e.g. using our algorithm without abstraction or CheckFence[15].

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

The best fit we have found to those criteria were well-known mutual exclusion (mutex) algorithms and synchronization barrier algorithms.

- Dekker’s Algorithm [25]. To evaluate both inference and verification we used two versions:
  - Dek0: has no added fences and is incorrect under the PSO memory model.
  - Dek2: has two added fences and is known to be correct.
- Peterson’s Algorithm [63], in two versions, Pet0 and Pet2.
- A variation of Lamport’s Bakery [42] in two versions, Lam0 and Lam2. To make this algorithm finite-space we manually bounded the maximum ticket number at 2.
- CLH queue lock [51] in two versions, CLH0 and CLH2
- Centralized sense-reversing synchronization barrier [32] in two versions Sense0 and Sense1.

All algorithms were exercised in a loop by two concurrent processes ("repeated entry"), which is the most general (two process) client for those algorithms.
**Abstractions** In our experiments, we consider a range of abstract memory models, all of which are abstractions of the concrete PSO memory model:

- **Set**: a naive abstraction of the store buffer to a set, without any additional information such as recency.
- **$\Box C$**: the must-contain partial coherence abstraction with $k$ varying between 0 and 2.
- **$\Diamond C$**: the may-contain abstraction with $k$ varying between 0 and 2.

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

**Specification** It is typical to require that mutual exclusion primitives satisfy two criteria: (i) mutual exclusion and (ii) no-deadlock. Here, for the sake of simplicity, we focused only on the first property. That is, we require the primitives to only satisfy mutual exclusion. Unfortunately, under relaxed memory models, even this simple specification is not necessarily well-defined. The traditional definition of mutual exclusion states that it is impossible for two processes to simultaneously execute code within the critical section protected by the mutex. This requirement is not necessarily sufficient under relaxed models. Consider a PSO execution in which process 0 takes the mutex, writes a value $v$ to some shared variable $x$, and releases the mutex. Before the newly written value is flushed, process 1 proceeds to take the mutex and read $x$ (thus getting the old value). It can be argued that this situation should also be considered a violation of mutual exclusion, which leads to a different specification: A primitive satisfies mutual exclusion if all values written to shared memory within the critical section of a process become globally visible before any other process may enter the critical section. While under sequential consistency both definitions are equivalent, under relaxed models the latter definition is strictly stronger (that is, implies) the former definition. In our experiments, we chose to use the first (weaker) definition, as it allows more flexibility in placing fences, and therefore is a more suitable benchmark for fence inference techniques.

**Verification**

Tab. 6.3 shows the verification results produced by **blender** with three abstractions. The programs we used here are the ones we know to be cor-
rect under the concrete PSO semantics, that is, appropriate fences have been placed in advance. All verification runs completed within 30 seconds. Each entry in the table contains the total number of states explored (in thousands). A ● mark is placed if verification succeeded and a ○ mark if a spurious (abstract) counter-example was found. In two of the runs of CLH2, BLENDER exhausted the available memory, and thus we do not report the state-space size. However, in both of those cases an (abstract) counter-example was found before BLENDER ran out of memory.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>Set</th>
<th>$\square C_{k=0}$</th>
<th>$\square C_{k=1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sense1</td>
<td>●</td>
<td>0.6</td>
<td>1.7</td>
</tr>
<tr>
<td>Pet2</td>
<td>○</td>
<td>7.7</td>
<td>2.4</td>
</tr>
<tr>
<td>Dek2</td>
<td>○</td>
<td>9.7</td>
<td>4.5</td>
</tr>
<tr>
<td>Lam2</td>
<td>○</td>
<td>41.2</td>
<td>22.2</td>
</tr>
<tr>
<td>Fast3</td>
<td>○</td>
<td>22.2</td>
<td>16.4</td>
</tr>
<tr>
<td>CLH2</td>
<td>○</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 6.3: Verification results and number of states (in thousands)

Discussion

As Tab. 6.3 shows, none of the correct examples could be verified using the naive set abstraction, however all of them could be verified using $\square C$ with $k = 1$. Since verification of all examples successfully completed with $\square C$, there was no need to use the $\diamond C$ abstraction. The table also shows $\square C_{k=0}$ generated spurious counter-examples for CLH2 and Fast3 but not the other algorithms. When $k = 0$, the partial-coherence abstraction ($\square C_{k=0}$) reduces to the set abstraction with recency information. This is enough to verify the simpler algorithms, however it fails on the more complex ones.

The example of Lamport’s fast mutex is particularly interesting, as it demonstrates the type of executions possible with non SC-recoverable abstractions. Consider the code in Fig. 6.2. In this implementation a process can enter the critical section either along the fast path (the if condition in Line 14 is false) or along the slow path (the conditions is true). Under an abstract model with $k = 0$, the following execution is possible:

- Process 1 enters the critical section along the fast path, executes it, and runs until Line 28.
- Process 1 executes Line 28. At this point $S_1(y) = \{0\}$.
- Process 1 flushes $y$ non-destructively, using the \texttt{FLUSH-SN} rule. Now $G(y) = 0$.
- Process 2 enters the critical section. Since $G(y) = 0$ it enters along the fast-path setting $y = 2$ in the process. This is flushed destructively using the \texttt{FLUSH-SD} rule. At this point $G(y) = 2$, $S_1(y) = \{0\}$, $S_2(y) = \emptyset$.
- Process 1 resumes. It first performs a flush of $y$, setting $G(y) = 0$. Then it proceeds to also enter the critical section, using the fast path. This execution relies on the fact $p_1$ only stored the value 0 to $y$ once, but this store is flushed twice. In effect, $p_2$ observed this store as if it happened \textit{before} its own, and $p_1$ observed it as if it happened \textit{after} the store of $p_2$. This coherence violation would have been prevented if we kept more information in the content of the buffer, by using $k > 0$. Indeed, with $k = 1$, \textit{Fast3} passes verification.

1 { 
2 start: 
3 store b[i] \leftarrow 1; 
4 store x \leftarrow i; 
5 load local\_y \leftarrow y; 
6 if (local\_y \neq 0) { 
7 store b[i] \leftarrow 0; 
8 while (local\_y \neq 0) 
9 load local\_y \leftarrow y; 
10 goto start; 
11 } 
12 store y \leftarrow i; 
13 load local\_x \leftarrow x; 
14 if (x \neq i) { 
15 store b[i] \leftarrow 0; 
16 do { 
17 load other\_b \leftarrow b[1-i]; 
18 } while (other\_b \neq 0); 
19 load local\_y \leftarrow y; 
20 if (local\_y \neq i) 
21 { 
22 while (local\_y \neq 0) 
23 load local\_y \leftarrow y; 
24 goto start; 
25 } 
26 } 
27 //Critical Section 
28 store y \leftarrow 0; 
29 store b[i] \leftarrow 0; 
30 } 

Figure 6.2: A version of Lamport’s fast mutex algorithm for 2 processors. The code given is for process $i$

Inference

In Tab. 6.4 we show the state-space size and inference results for 5 of the under-fenced implementations. A mark of \textbullet{} means the optimal fences were inferred, \textcircled{1} means that sub-optimal fences were inferred, and \textcircled{0} means that \texttt{BLENDER} was unable to infer fences as according to the analysis any fence
placement would leave the program incorrect. \( M \) appears if \textsc{blender} ran out of memory.

\begin{table}
\begin{center}
\begin{tabular}{l|ccc|ccc}
\hline
& \( \square C_{k=0} \) & \( \square C_{k=1} \) & \( \Diamond C_{k=0} \) & \( \Diamond C_{k=1} \) & \( \Diamond C_{k=2} \) \\
\hline
\textit{Sensel0} & \( \bullet \) & 57.6 & \( \bullet \) & 58.6 & \( \bullet \) & 31.9 & \( \bullet \) & 44.3 & \( \bullet \) & 6.1 \\
\textit{Pet0} & \( \bullet \) & 69.2 & \( \bullet \) & 524.5 & \( \bullet \) & 25.3 & \( \bullet \) & 244.7 & \( \bullet \) & 1121.2 \\
\textit{Dek0} & \( \bullet \) & 424.2 & \( \bullet \) & 3258.1 & \( \bullet \) & 16.5 & \( \bullet \) & 3581.1 & \( \bullet \) & 23590.0 \\
\textit{Lam0} & \( \bullet \) & \( \bullet \) & 421.0 & \( \bullet \) & 4045.9 & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) \\
\textit{Fast0} & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & 220.6 & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) \\
\textit{Fast1a} & \( \bullet \) & 139.0 & \( \bullet \) & 105.2 & \( \bullet \) & 33.0 & \( \bullet \) & 88.1 & \( \bullet \) & 41.9 \\
\textit{Fast1b} & \( \bullet \) & 832.9 & \( \bullet \) & 972.3 & \( \bullet \) & 78.6 & \( \bullet \) & 501.6 & \( \bullet \) & 878.4 \\
\textit{Fast1c} & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & 110.4 & \( \bullet \) & 1173.4 & \( \bullet \) & 1858.1 \\
\textit{CLH0} & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) & \( \bullet \) \\
\hline
\end{tabular}
\end{center}
\caption{Inference results and number of states (in thousands)}
\end{table}

\section*{Discussion}

Initially, we used \textsc{blender} to perform fence inference with abstractions \( \square C_{k=0} \) and \( \square C_{k=1} \). However, \textsc{blender} ran out of memory for \textit{Lam0}, \textit{Fast0}, and \textit{Fast1c}. Using the may-contain abstraction \( \Diamond C_{k=0} \) enabled us to run the inference algorithm for both \textit{Lam0} and \textit{Fast1c} and obtain a sound fence placement for both. Furthermore, despite the loss of precision in the \( \Diamond C \) abstraction, in both cases the inferred fences are not trivial.

\textbf{Peterson’s Algorithm} Our results for Peterson’s algorithm demonstrate the inherent trade-offs between inference optimality and abstraction precision:

- With the \( \square C \) abstraction \textsc{blender} was able to infer the optimal fence placement with \( k = 0 \). With the \( \Diamond C \) abstraction it required \( k = 2 \) and a much larger state-space.
- With the \( \Diamond C_{k=0} \) abstraction we can produce a smaller state space but the result is suboptimal: 3 fences are required instead of 2. In addition to the two fences shown in Fig. 5.9, another fence, immediately after the store in line 14, is inferred.

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

\textbf{Dekker’s Algorithm} Dekker’s algorithm was the only algorithm for which we used both \textsc{fender} and \textsc{blender} for fence inference. It is well known that Dekker’s algorithm requires a fence in the entry section and a fence at the


end of the critical section. Using FENDER, in the RLX framework, both fences were inferred for RMO and PSO. Under TSO, only a single fence was inferred. This is consistent with the reference placement in Appendix J of the SPARC Architecture Manual [36]. However, while we did infer a safe fence placement, the process through which this was achieved was not a priori sound. This was due to two reasons:

- The client for Dekker’s algorithm used in our earlier experiments was a “single-entry” client: the two processes attempted to enter the critical section exactly once, as opposed to the most general client which does so in a loop.
- Even with a single-entry client Dekker’s algorithm contains potentially unbounded spinning. Therefore, to generate a finite transition system we had to add arbitrary bounds on the length of the executions buffers (as indicated in Tab. 6.1).

By applying partial-coherence abstraction we overcame both issues, and used BLENDER for sound fence inference with the most general repeated-entry client. The result was, again, the optimal 2-fence placement, with the same trade-offs as described for Peterson’s algorithm.

**Lamport’s Fast Mutex** For both Fast0 and Sense0, we experienced a loss of precision when using a too small $k$ value. In the case of Fast0, the inference algorithm reported the program as unfixable when using $C_k=0$. This is due to the fact the counter-example presented for Fast3 under this abstract model cannot be fixed with any number of fences. Unfortunately, BLENDER was unable to build the state-space of Fast0 under $C_k=1$. Thus, we’ve run a complementary set of experiments in which 1 of the 3 required fences was placed. The 3 versions of Lamport’s fast mutex (Fig. 6.2) we have ran had a single fence inserted: (i) between lines 4 and 5 (Fast1a), (ii) between lines 12 and 13 (Fast1b), (iii) between lines 28 and 29 (Fast1c). As expected, for all 3 programs, when running under $C_k=0$ the program was unfixable. However, in all cases we were able to infer a correct fence placement using $C_k=1$. Further, for Fast1a and Fast1b the optimal placement of the other fences was found when using $C_k=2$. For Fast1c even with $k = 2$ the placement was still suboptimal. This is another example of the interplay between placed fences and precision of the abstraction. Even though for Fast1c we could not infer the optimal fence placement using $C_k=1$, had we placed these fences manually, this abstraction could be used to verify them.
Chapter 7

Related Work

In this chapter, we survey the related work in the fields of verification and memory fence inference under relaxed memory models. In the latter part of the chapter, we also mention some other synthesis techniques which may be applicable to fence inference, even if they have not been used for this purpose.

Formalization of Memory Models

While this work does not deal directly with formalizing hardware-level relaxed memory models, it heavily relies on the large body of work done in the field. The very first paper to deal with memory models was the seminal paper by Lamport [43] that defined the concept of Sequential Consistency. The theoretical interest in weaker forms of consistency resulted in the definitions of weaker consistency models, such as Weak Consistency [26] and Release Consistency [30]. On the more practical side, the actual memory models used by processors have been defined in the architecture manuals of those processors, e.g. the SPARC [36], PowerPC [35] and the Intel x86 architecture manuals. Unfortunately, the early editions of those manuals tended to give very informal specifications of the models, which rendered them near to useless (a good overview of the shortcomings of various Intel manuals can be found in Sarkar et al. [69]). Efforts to formalize those models have begun in the mid 90s — for example Park & Dill’s operational formalization of SPARC RMO [62], and work by Corella et al. [22] and later Attiya et al. [1] on formalizing PowerPC. A very good (although somewhat
dated) introduction to the mid-90s state of the art in this area can be found in a tutorial by Adve & Gharachorloo from 1995 [3]. Concurrently, research was carried out on frameworks that enable easier formalization of various models. Two of those frameworks, which we used to help us construct our sample framework RLX are CRF (Arvind et al. [71]) and UMM (Yang et al. [81]). More recently, due to renewed interest in relaxed memory models, a new effort of systematic formalization resulted in the publication of papers fully formalizing the x86 (Owens et al. [61]) and most recently PowerPC (Sarkar et al. [68]) memory models. In addition to hardware-level models, a large body of work exists on the formalization of language-level (e.g. the Java [53] and C++11 [10]) memory models. A modern perspective on the challenges of language-level models can be found in a recent paper by Boehm & Adve [2].

Verification under RMM

The verification problem under relaxed memory models clearly appears to be hard. The first to show it is formally hard were Atig et al. [5]. Their paper shows reachability for SPARC RMO is undecidable, while for SPARC PSO and TSO it is decidable but has non-primitive-recursive complexity. This validates our conclusion that to obtain a precise and sound verification (and inference) procedure, even for SPARC TSO, abstraction is a necessary step. However, even algorithms that imprecise or not completely sound can often be very useful, and several of those have been developed.

Specification-Agnostic Verification

In [16] and [17] Burckhardt et al. and Burnim et al., respectively, present algorithms that, based on exploring only sequentially consistent executions, can find violations of sequential consistency under the TSO and PSO models. Unfortunately, this is often needlessly conservative, as not every violation of sequential consistency is also a violation of the real high-level specification.

Under-approximation

A different way to deal with the problem is to allow the algorithm be unsound by introducing arbitrary bounds.
Explicit-state model-checking for the SPARC RMO model, performed in a fashion similar to our implementation has appeared in work by Park & Dill [62] and a decade later Jonsson [38]. However, neither of them used it for fence inference.

In [14, 15] Burckhardt et al. present the CheckFence tool which takes a different approach. Instead of working with operational memory models and explicit model-checking, they convert programs into a form that can be checked against an axiomatic model specification. This technique still suffers from the same limitation as the explicit techniques - it must unroll loops at a preprocessing stage, so it cannot verify programs that contain unbounded spinning.

Finally, in [18], Burnim et al. describe yet another approach. Instead of performing bounded model-checking, they only explore part of the transition system (that is, they perform testing) of the program. While this approach improves scalability, this is even farther away from sound verification.

Data-Race Freedom

It is natural that some classes of programs can be easier to treat than the most general case. For verification under relaxed memory models, the class usually under discussion is that of program that do not contain data races — in other words, they are “data-race free” (DRF). According to Saraswat et al. [67], “The fundamental property of memory models” is that a program that is both correct and data-race free under the SC memory model will remain correct under the relaxed model. And indeed, all practical memory models satisfy this property (also known as the “DRF Guarantee”). This implies that given a correct SC program, if we can prove that it is DRF, we know it is also correct under any memory model of interest.

In [60], Owens studied a special case of the theorem for the TSO memory model. As he shows, to guarantee correctness under TSO one needs only to prove the program’s sequentially consistent executions satisfy “triangular-race freedom” (TRF), a property weaker than general DRF.

Unfortunately, many interesting programs contain (intentional) data races, and even triangular races. This applies in particular to high-performance concurrency primitives and non-blocking data structures of the type dealt with in this thesis.
Effect Mitigation

Another approach to the problem is that of effect mitigation. Instead of trying to reduce the number of fences in a program, it is possible to reduce the performance hit caused by each fence, thus avoiding the issue. For example, in [8] Blundell et al. propose “InvisiFence”, a set of CPU architectural improvements that minimizes the performance impact of fences. Another effort in this direction is work by von Praun et al. ([74, 80]). Whether this works in practice, “outside the lab” is a hotly-debated topic. We believe this approach is complementary to our work — while some practical reduction in the performance impact of fences is definitely possible, it does not alleviate the need for judicious fence placement.

Abstraction

Using bounded structures to represent potentially unbounded fifo queues is not, in itself, a new idea. In [46] Linden & Wolper use automata as symbolic representation of store buffers in the TSO memory model. In [47], which was published almost concurrently with [41], they extend this technique to fence inference. Unlike our work, their approach uses an acceleration technique that does not guarantee termination. Furthermore, their automata-based representation preserves redundant information and, as noted by the authors themselves in [46], ends up being too expensive to be of practical interest.

Since store buffers are similar to FIFO channels in communicating FSMs (CFMSs), it is tempting to employ techniques from CFMSs in the context of store-buffer based memory models. These techniques include algorithms based on symbolic representation of channel content (e.g. work by Boigelot et al. [11] which served as the basis for the previously mentioned work by Linden), and conservative abstractions for FIFO channels (Le Gall et al. [28]). Abstraction of FIFO channels, as presented in [28], is similar in spirit to our approach in that it guarantees termination by using over-approximation. However, their abstraction preserves a slightly different kind of information than the information required for reasoning about store buffers. They use a regular abstraction of queue content and an expensive widening operator to establish the correct usage of protocols. This is more than what is required in our setting in terms of characterization of buffer content, and often less than needed in terms of recency informa-
Fence Inference/Synthesis

Several automated techniques to place memory fences in concurrent programs have been developed over the years.

Enforcing equivalence to SC

A large body of work dating back to the late 1980s relies on the concepts of delay set analysis of Shasha & Snir [70] for reasoning about relaxed memory models. This analysis enables one to find all potential conflicts (more or less equivalent to data races), and place fences accordingly. A fence inference scheme based on delay set analysis was successfully implemented in the “Pensieve” Java compiler [45, 27, 72], which can effectively process large amounts of code. However, as pointed out earlier in this chapter, a violation of SC does not necessarily cause a violation of any high-level properties. Thus those algorithms are often needlessly conservative. Our approach, on the other hand, uses a high-level specification and allows a trade-off between performance and optimality of the solution.

Syntactic exploration

Another possible approach to fence inference is to use a verification tool combined with syntactic exploration. To utilize a verification tool (e.g. CheckFence) for inference, the programmer uses an iterative process: she starts with an initial fence placement and if the placement is incorrect, she has to examine the (non-trivial) counterexample from the tool, understand the cause of error and attempt to fix it by placing a memory fence at some program location. It is also possible to use the tool by starting with a very conservative placement and choose fences to remove until a counterexample is encountered. This process, while simple, may easily lead to a “local minimum” and an inefficient placement. A technique similar to this (albeit significantly more sophisticated) has been previously used in other contexts, for example by Vechev et al. [77, 76].

In [47], Linden & Wolper use automate this approach, using the technique described in [46] as the underlying verification tool. However, their
tool still suffers from the same problem - it does not necessarily provide a globally optimal solution.

**Graph-based approach**

In [34], Hyunh et al. presented the tool *mmchecker*. This tool is, similarly to our own, based on explicit state model-checking and uses a naive approach to fence inference. In their work, Hyunh et al. formulate the problem of fence inference as a minimum cut on the reachability graph. While the result produced by solving for a minimum cut is sound, it is often suboptimal. The key problem stems from the lack of one-to-one correspondence between fences and removed edges. First, the insertion of a single fence has the potential effect of removing many edges from the graph. So it is possible that a cut produced by a single fence will be much larger in terms of edges than that produced by multiple fences. [34] attempts to compensate for this by using a weighing scheme, however this weighing does not provide the desired result. Worse yet, the algorithm assumes that there exists a single fence that can be used to remove any given edge. This assumption may cause a linear number of fences to be generated, when a single fence is sufficient.

**Synthesis**

There are many previously published approaches to program synthesis, and two of them are closely related to the part of our work described in Chapter 4, even if they do not deal with memory fences.

This thesis is a direct continuation of earlier work by Vechev, Yahav and Yorsh [78, 79]. In their work on “Abstraction-Guided Synthesis” (AGS) [79] Vechev et al. used ideas similar to those presented in Chapter 4 to infer minimal synchronization regions in multi-process programs. The main difference between AGS and this work is that the algorithm given in [79] is trace-based, while the algorithm given in Chapter 4 is state-based. This means our algorithm scales much better to larger transition systems. In fact, the algorithm was developed precisely to address the scalability problems of [79].

Another technique, pioneered by Jobstmann et al. [37] is program repair through the use of games. Very informally, the basic idea behind this tech-
nique is to represent the transition system as a game between the environment and the system, where the system attempts to steer execution towards satisfying the specification, while the environment attempts the opposite. In the case of safety specifications, the result is a safety game: the system wins the game if it can avoid entering an error state, while the environment wins if it force the system into such a state. The choices the environment is allowed to make during the game are given by the non-determinism inherent in the execution. For example, in concurrent programs the environment may decide which process to schedule next. The choices of the system are constrained by the repair tool’s user. As we are interested in program repair, as opposed to program synthesis, the system is assumed to be mostly complete and deterministic. However, it is allowed to make some non-deterministic choices - for example, it may be allowed to decide which value to assign to a variable at a given point in the execution. It is the synthesizer’s role to find a winning strategy for the system, and if such a strategy is found convert it into a fix for the program.

It is easy to adapt this technique to our setting. Our semantics are inherently non-deterministic, even for a single processor. So, the environment, in addition to choosing which process should be scheduled, may also choose what kind of action to perform (e.g. whether it should flush the store buffer or execute the next statement for that process). Conversely, the system may, at any point of the execution, decide to force the next operation to be a fence. Unfortunately, while this adaptation will produce a safe solution, it can not guarantee optimality of any kind. The algorithm of Jobstmann et al. will select some arbitrary memoryless winning strategy, which will not, in general, result in the minimal number of fences being inserted into the program.
Chapter 8

Future Work

I like the dreams of the future better than the history of the past. — Thomas Jefferson

In this thesis we presented a novel fence inference algorithm and demonstrated its practical effectiveness by evaluating it on state-of-the-art concurrent algorithms. We also presented a technique that allowed us to extend the applicability of this algorithm to a larger family of programs than those directly supported by the algorithm. Our approach is based on abstract interpretation, and its technical core is a family of partial-coherence abstractions that provide a (parametric) bounded representation for potentially unbounded store buffers. These abstractions enable us to automatically verify and infer fences for concurrent programs without worrying about the size of the underlying store buffers.

The main limitation of our approach is the set of programs to which it is applicable. In particular, even using partial-coherence abstractions, we require the input program’s transition system to be finite and explicitly constructible. Unfortunately, this is not the case for many interesting programs, and we see solving this limitation as the main direction for future work.

Under-approximation

One way to deal with the infinite-state (or large) systems is to under-approximate the transition system: instead of exploring the entire transition system, we may limit exploration to only a subset of it. For example, we can use dynamic methods to produce a (non-comprehensive) set of error
traces. We can then use this set of traces to generate and solve a constraint. Preliminary work in this direction, by Liu et al., has been recently accepted for publication [48]. While this method is not sound it enables the extension of our algorithm to much larger programs - for example, Liu et al. used it to infer fences in a lock-free memory allocator due to Maged Michael [55]. The performance of dynamic methods depends heavily on “smart” selection of error traces, and we believe that much more progress can be made in this direction.

Combination of Abstractions

Another, arguably better, way to solve the scalability problem is to combine our partial-coherence abstractions with existing abstract interpretation techniques. Given a program which is infinite-state even under SC, it is tempting to consider a two-step technique: first apply a “standard” abstraction to make the program finite-state, and then use partial-coherence abstraction to keep it finite-state under a relaxed model. Unfortunately, even without partial-coherence abstractions, the “standard” abstraction and the memory model may have non-trivial interactions.

The simplest case to handle is the case of value abstractions. In Chapter 5, we assumed the program’s variables take values from some finite set \( D \). In case \( D \) is infinite, it is often useful to interpret the program over some finite set \( D^\# \) instead. In this case each value \( v \in D \) is represented by some abstract \( \alpha(v) \in D^\# \) (hence the name “value abstraction”), and the program semantics are appropriately changed to support expression evaluation over the new domain of values. A classical example, due to Cousot, is the “sign abstraction” of integers, where \( D = \mathbb{N}, D^\# = \{\bot, +, -, 0\} \) and for \( v \in \mathbb{N} \), \( \alpha(v) \) is determined according to the sign of \( v \). The required changes to program semantics are then mostly restricted to expression evaluation. For example, the value of the expression \( x + y \) is now evaluated according to the law of signs (if both \( x \) and \( y \) are positive then the result is positive, etc.) For value abstractions, the relaxed state representation is a simple adaptation of the representation from Chapter 5. In such an adaptation, the store buffers, instead of containing concrete values from \( D \), contain abstract values from \( D^\# \). Since there is very little interaction between expression evaluation (which is done over local variables) and the core rules of the
memory model, the relaxed semantics are also easy to adapt. Applying a partial-coherence reduction is then trivial: the abstract state representation contains abstract buffers of abstract values.

Other types of abstractions present harder challenges. Two common types of useful abstractions are predicate abstractions [31] and heap abstractions (e.g. shape abstraction [66]). Without going into details, the main idea behind predicate abstraction is that instead of keeping track of precise variable evaluations, the abstract semantics only keep track of certain predicates over those variables. For instance, while the precise values of variables $x$ and $y$ may be lost, the information that $x < y$ is retained. Since such predicates usually involve several variables, it is unclear how they can be combined with per-variable store buffers.

The issues with shape abstraction are similar. Shape abstraction is used when a program is not finite-space due to the use of an unbounded number of heap locations — as a trivial example consider a program that allocates memory in an infinite loop. Very informally, shape abstraction does not preserve full information about all reachable heap locations. Rather, it preserves some information about the general “shape” of the heap. This means that the semantics used with shape abstraction will often perform steps that alter the shape of the heap. Such a step may, for example, combine several heap locations into one abstract location, or perform partial concretization and “factor out” a concrete location from an abstract one. Again, it is unclear how to combine this with PSO store buffers that need to be maintained per heap location.

We believe that analysis of programs under relaxed memory models can only become truly useful if it can be effectively combined with existing abstract interpretation techniques. Therefore we see resolving the problems described above as our most important direction for future work.
Bibliography


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


[60] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, 2010.


כאמור אל פרmalink, אפישר לברר שמודל זכר מופשט הוא "קורות מלעיים" של מודל זכר לסירוגין, במענה הא.keys: כמדרשת開放 alt ת mạchণית תocha מודל אינפֶרור תוחזק開放 alt ת mạchणית מופשט (אף אל בכרח להפחית)._transportation_method אינפֶרור תוחזק開放 alt ת mạchणית המופשט עיקרית ות mạchणית מודל בעלת רֵעַיָּב מרב חומבי סופי תוחזק開放 alt ת mạchणית מודל המופשט open up_start בפעולה עם אינפֶרור תוחזק開放 alt ת mạchणית מודל המופשט open up_end עב하다 סדרות הדגシי עבדת מзвук סביר סופי עבדת תוחזק開放 alt ת mạchणית המופשט עלי.ידי שומש במדלימום סולם, וה.slimים לשרטוטם באלגוריתם שפיקוח על יובל תוחזק開放 alt ת mạchणית המופשט open up_start המנהט על מטעם סופי במדל עכבר סדרות. תוחזק開放 alt ת mạchণית המופשט open up_end

ל시스ם, الهندורסט אתניקיה על warto וניה.

- מטפחתו של מודל זכר מופשטープמורפהת ממדל סופי של תוחזק開放 alt ת mạchणית המופשט open up_start
  • עבורד מבחר גזז של מודל ירכיה.
- אינפֶרור תוחזק開放 alt ת mạchণית מודל הапрומיסית של דרורת זכר בתוכגיית מקבילית open up_end
  • מבחר מזרחי מסיבים סופיים.
- הב- TempData של כל המمعنى את מודל הליגר מופשייס המלגיית_Generic open up_start
  • תפריך יסוד את איב-TempData של ליגר שㇼחרתה שمشاريع לשרטוטם באלגוריתם open up_end
cדי ליצא הגרות זכר במדל אינפֶרור תוחזק開放 alt ת mạchণית מקבילית מדרדרי, ובפרטי
  • מבני תרגום מקבילים מונגולים-שניבור פ責יס.
null
typedef struct
{
    long size;
    int *ap;
} item_t;

long top, bottom;
item_t *wsq;

void push(int task) {
    long b = bottom;
    long t = top;
    item_t* q = wsq;
    if (b−t ≥ q->size−1) {  
        q = expand();  
    }
    q->ap[b % q->size] = task;
    fence("store−store");
    bottom = b + 1;
}

int take() {
    long b = bottom − 1;
    item_t* q = wsq;
    fence("store−load");
    long t = top;
    if (b < t) {
        bottom = t;
        return EMPTY;
    }
    task = q->ap[b % q->size];
    if (b > t)
        return task;
    if (!CAS(&top, t, t + 1))
        return ABORT;
    bottom = t + 1;
    return task;
}

item_t* expand() {
    int newsize = wsq->size * 2;
    int* newitems = (int*) malloc(newsize*sizeof(int));
    item_t* newq = (item_t*) malloc(sizeof(item_t));
    for (long i = top; i < bottom; i++) {
        newitems[i % newsize] = wsq->ap[i % wsq->size];
    }
    newq->size = newsize;
    newq->ap = newitems;
    fence("store−store");
    wsq = newq;
    return newq;
}
This page seems to contain a table and some text in Hebrew. Here is the table in Markdown format:

<table>
<thead>
<tr>
<th>Locations</th>
<th>Effect of Reorder</th>
<th>Needed Fence</th>
</tr>
</thead>
<tbody>
<tr>
<td>push:8:9</td>
<td>steal() returns phantom item</td>
<td>store-store</td>
</tr>
<tr>
<td>take:4:5</td>
<td>lost items</td>
<td>store-load</td>
</tr>
<tr>
<td>steal:2:3</td>
<td>lost items</td>
<td>load-load</td>
</tr>
<tr>
<td>steal:3:4</td>
<td>array access out of bounds</td>
<td>load-load</td>
</tr>
<tr>
<td>steal:7:8</td>
<td>lost items</td>
<td>store-load</td>
</tr>
<tr>
<td>expand:9:10</td>
<td>steal() returns phantom item</td>
<td>store-store</td>
</tr>
</tbody>
</table>

The table likely represents some form of analysis or results from a study or research. The text in Hebrew seems to provide additional context or explanation related to the table.
התקציר

בשנת 1979, נ ecxאפ המס摭, "אקז ליזיצ מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז מנקז
انتشهد המחבר שים שלמה על התמצאותו kỹונטיות בפניהם.

לא להיות לטעינו למפרק את המחבר שלום על תดวงות הנספים והניצבות בים.

הלכודתי.
퇀ור בֵּנְג'וֹת תַּחַת מְדְלִילִים בֶּקרֶה

תִּיוֹבְר על מַחְקֶר

לְשׁוֹנַיָּר חֲלֵק שֶל הַדִּרְשָׁהּ לַקָּבֵלָת הַזְּהָהּּ

מִמְסֶר לְמִדְעָמ בָּמְדִיעַ הַמַּחְשִׁיב

מיכָּאל קָוְרֵשְׁטיוּן

חוגֶשׁ פַּסְגַּנָּה הָטָכָנוֹת - מָמוֹן טָכָנוֹנִי לְיִשְׂרָאֵל

טבְּנַת הַדִּישֵׁנָּב - חִיפָּה

פֶּבְרָרָא 2012
שימור נוכחות תחת מודל גיבורי הלשון

ميיכאל קופרטיין