Verification of Software under Relaxed Memory

CARL LEONARDSSON
Dissertation presented at Uppsala University to be publicly examined in 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Wednesday, 7 September 2016 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Viktor Vafeiadis (Max Planck Institute for Software Systems).

Abstract

The work covered in this thesis concerns automatic analysis of correctness of parallel programs running under relaxed memory models.

When a parallel program is compiled and executed on a modern architecture, various optimizations may cause it to behave in unexpected ways. In particular, accesses to the shared memory may appear in the execution in the opposite order to how they appear in the control flow of the original program source code. The memory model determines which memory accesses can be reordered in a program on a given system. Any memory model that allows some observable memory access reordering is called a relaxed memory model. The reorderings may cause bugs and make the production of parallel programs more difficult.

In this work, we consider three main approaches to analysis of correctness of programs running under relaxed memory models. An exact analysis for finite state programs running under the TSO memory model (Paper I). This technique is based on the well quasi ordering framework. An over-approximate analysis for integer programs running under TSO (Paper II), based on predicate abstraction combined with a buffer abstraction. Two under-approximate analysis techniques for programs running under the TSO, PSO or POWER memory models (Papers III and IV). The latter two techniques are based on stateless model checking and dynamic partial order reduction.

In addition to determining whether a program is correct under a given memory model, the problem of automatic fence synthesis is also considered. A memory fence is an instruction that can be inserted into a program in order to locally disable some memory access reorderings. The fence synthesis problem is the problem of automatically inferring a minimal set of memory fences which restores sufficient order in a given program to ensure its correctness.

Keywords: verification, relaxed memory models

Carl Leonardsson, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Carl Leonardsson 2016

ISSN 1651-6214
ISBN 978-91-554-9616-6
urn:nbn:se:uu:diva-297201 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-297201)
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I  Counter-Example Guided Fence Insertion under TSO [5]
    Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, Ahmed Rezine
    TACAS 2012

II Automatic Fence Insertion in Integer Programs via Predicate Abstraction [4]
    Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, Ahmed Rezine
    SAS 2012

III Stateless Model Checking for TSO and PSO [2, 3]
    Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson,
    Konstantinos Sagonas
    TACAS 2015
    Technical report accepted for publication in Acta Informatica

IV Stateless Model Checking for POWER [6, 7]
    Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson
    To appear in CAV 2016

Reprints were made with permission from the publishers.
## Contents

1 Introduction .................................................................................................................. 7

2 Concurrent Programs .................................................................................................. 11
  2.1 Preliminaries ........................................................................................................ 11
  2.2 Program Model .................................................................................................... 12
    2.2.1 Language ....................................................................................................... 12
    2.2.2 Program Objects ......................................................................................... 15
    2.2.3 Semantics under Sequential Consistency ..................................................... 16

3 Relaxed Memory ......................................................................................................... 21
  3.1 Relaxed Memory: The Why and How ................................................................. 21
    3.1.1 Memory Access Reordering ......................................................................... 21
    3.1.2 Shasha-Snir Traces ..................................................................................... 25
    3.1.3 Optimizations and Consistency Relaxation ................................................ 26
    3.1.4 Restoring Order ......................................................................................... 31
  3.2 TSO and PSO ....................................................................................................... 36
    3.2.1 Semantics of TSO ....................................................................................... 36
    3.2.2 Semantics of PSO ....................................................................................... 40
  3.3 POWER ................................................................................................................. 44
    3.3.1 Intuition of POWER .................................................................................... 45
    3.3.2 Shasha-Snir Traces: Formalization and Derived Relations ......................... 48
    3.3.3 The POWER Memory Model ................................................................. 52
    3.3.4 POWER Examples Explained ................................................................. 56

4 Stateless Model Checking ......................................................................................... 59
  4.1 Source-DPOR ....................................................................................................... 62
    4.1.1 Source-DPOR Algorithm .......................................................................... 66
  4.2 Chronological Traces for TSO and PSO ............................................................. 68
  4.3 RSMC for POWER ............................................................................................. 72
    4.3.1 Operational POWER ................................................................................. 73
    4.3.2 RSMC ....................................................................................................... 76

5 Fence Synthesis ....................................................................................................... 79
  5.1 A Fence Insertion Algorithm ............................................................................. 81
  5.2 Witness Analysis ................................................................................................. 85
  5.3 Proof of Correctness ......................................................................................... 87
    5.3.1 Termination .............................................................................................. 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2</td>
<td>Invariant</td>
<td>88</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Soundness</td>
<td>88</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Completeness</td>
<td>89</td>
</tr>
<tr>
<td>5.4</td>
<td>Optimizations and Variations</td>
<td>89</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Optimizing the Witness Analysis</td>
<td>89</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Approximate Reachability Engines</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>93</td>
</tr>
<tr>
<td>7</td>
<td>Summary in Swedish: Verifiering av program under svaga minnesmodeller</td>
<td>95</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>My Contributions</td>
<td></td>
<td>102</td>
</tr>
</tbody>
</table>
In this thesis, we will examine a number of techniques for automatic verification and testing of parallel programs running under relaxed memory models.

As processor clock speed increases slow down and multi-core chips are becoming ubiquitous [52], programmers increasingly have to employ parallel programming to improve performance and scalability of their applications. However, parallel programming for high performance is difficult, and bugs can easily be introduced by the subtle interaction between different threads [24]. Furthermore, the debugging process is made more difficult in a parallel setting by the fact that bugs may occur very unpredictably, based on the thread scheduling, even when a program is executed multiple times on the same input. In the context of parallel programs that strive to further improve performance by implementing lock-free algorithms, the difficulties are even greater, due to the presence of so called relaxed memory. The term relaxed memory refers to the phenomenon of optimizations in hardware, compilers, virtual machines, etc., which may cause memory accesses by one thread to appear to another thread in an order which is opposite to how they were issued in the program-flow of the original source code. Such memory access reorderings make it even more difficult to formulate and reason about algorithms where different threads communicate by writing to and reading from the shared memory. The risk of errors then also increases. It is therefore important to have automatic tools that can help with verification and testing, in order to find and correct bugs early, and in the best case establish some certainty about the correctness of the programs.

For such tools to be applicable, it is necessary to provide a specification for the program. In fact, without a specification we cannot even reason about what it means for the program to be correct or incorrect. Two main classes of specifications are safety specifications and liveness specifications. Specifications in the former class make statements about the behavior of finite program executions. For example, a safety specification might declare that the program is never in a state where a division by zero occurs, or in a state where traffic
lights in a crossing show green simultaneously for two perpendicular roads. Specification in the latter class reason about the behavior of infinite program executions. For example, a liveness specification might state that the program cannot run forever without delivering a result, or that when a resource is shared between different processes it will be allocated to different processes at different times, rather than always being allocated to the same process.

Specifications need to be formalized in order to be usable by automatic verification or testing tools. This is necessary because the tools must be able to tell with certainty whether a given behavior is permitted, or constitutes a bug. The specification for a program may be constructed by a human, or automatically generated. A manually constructed specification can be better tuned to the intended context of the program. On the other hand, manual construction of specifications is difficult and error prone, much like the development of the program itself. For example, an automatically generated specification might contain criteria such as that the program should never divide by zero, never crash etc., while manual specification is necessary to capture context-specific criteria such as which color combinations are safe in a traffic light, or what is considered a useful result of a computation. In this thesis, we will focus on safety specifications (as opposed to liveness specifications), and typically assume that the specifications are provided by the user.

In cases where memory access reorderings by the relaxed memory cause errors, it is often necessary to be able to enforce the intended order between some crucial memory accesses. To make this possible, architects of hardware or compilers with relaxed memory typically provide special instructions with no effect other than enforcing order between surrounding memory accesses. These special instructions are known as fences or memory barriers. Such fences, when placed in the correct locations, can enforce the ordering between the crucial memory accesses, and restore correctness of the program. However, since fences locally disable the hardware/compiler optimizations which cause the memory access reorderings, they come at a performance penalty. Therefore it is important for programmers to carefully determine where to put fences, so that the program is correct, but the performance is not destroyed by excessive fences. So called automatic fence insertion techniques can help with the task of finding good fence placements. Automatic fence insertion techniques are techniques which, for a given program and specification determine where fences are needed to ensure that the program satisfies the specification. Fence insertion is typically based on some underlying automatic verification or systematic testing technique.

There are many approaches to automatic verification and systematic testing for programs in general, for parallel programs, and even for parallel programs under relaxed memory. Broadly, they can be partitioned into approaches that analyze program behaviors precisely, over-approximatively or under-approximatively. A precise technique covers precisely the behaviors of a program. If such a technique determines that the program is correct (with respect to some
given specification), then the program is correct. If the technique determines that the program contains errors, then those errors are real errors that could manifest in an actual execution of the program.

In contrast, an over-approximative technique approximates the behaviors of a program, in such a way that it covers all the actual behaviors of the program, but also possibly some additional spurious behaviors. This is typically done because the abstraction allows to efficiently generalize over, and deal with a huge number of actual behaviors. If an over-approximative technique determines that the program satisfies its safety specification, then the actual program is guaranteed to satisfy the specification, since any violation would have been discovered by the over-approximative technique. On the other hand, an over-approximative technique may find errors among the spurious behaviors, which are not present in the actual program.

An under-approximative technique approximates the behaviors of a program by covering a subset of its behaviors. An under-approximative technique may fail to discover real errors, but any error that it discovers is guaranteed to be a real error of the program. Simply running the program once, and observing its behavior can be considered a simple, random under-approximative analysis.

In this thesis, we will encounter examples of all the three approaches. In particular, Paper I describes a precise analysis. Paper II uses over-approximative techniques, paired with counter-example guided abstraction refinement. Papers III and IV describe under-approximations.

Outline of the Text.
The introductory chapters of this text are intended to provide the background necessary to understand the attached articles.

In Chapter 2, we introduce prerequisite formalisms for the rest of the text. We introduce a simple programming language, and define its semantics under the commonly known sequential consistency model. In Chapter 3, we examine what relaxed memory semantics are, and why they exist. We cover in detail the semantics of the relaxed memory models TSO, PSO and POWER. In Chapter 4, we examine stateless model checking. First we have a look at previous techniques for stateless model checking under sequential consistency. Then we briefly investigate the intuition behind my extensions of stateless model checking to relaxed memory, covered in detail in Papers III and IV. In Chapter 5, we examine how techniques for automatic verification and testing of programs under relaxed memory can be used as a foundation for automatically inferring memory fence placements that ensure the correctness of a program when executed under relaxed memory.

Brief Summary of the Papers
Paper I introduces a sound and complete fence insertion procedure for finite state programs running under TSO. It pairs a reachability analysis engine for
TSO with a counter-example guided fence insertion algorithm. The reachability algorithm is a development of the one published as a decidability proof in [16]. The challenge is that many programs induce infinite state spaces when executed under the TSO memory model, even if their state spaces under sequential consistency is finite. This is caused by the unbounded store buffers employed under TSO. The solution in Paper I uses the well-quasi ordered framework of [8] to enable a sound and complete backward reachability analysis even in such circumstances. The technique has been implemented in the tool Memorax, which is publicly available at [33].

Paper II suggests a technique for reachability analysis and fence insertion for programs under TSO, supporting variables with infinite integer data domains. The technique combines an over-approximate abstraction of the store buffers with predicate abstraction for the integer data values. Further, it employs a CEGAR loop which can refine either the buffer abstraction or the predicate abstraction as needed in order to address spurious errors.

Papers III and IV aim to develop more practically usable approaches to reachability analysis under relaxed memory. They adapt the technique of stateless model checking to the TSO and PSO (Paper III) and POWER (Paper IV) memory models. This allows an under-approximate analysis to target more realistic parallel test cases written in e.g. C/pthreads. The challenge here is to minimize the number of different program behaviors that are investigated, while still keeping the coverage extensive enough to detect errors. In Paper III we do this by applying dynamic partial order reduction techniques to a carefully designed representation of traces under TSO and PSO. In Paper IV we develop a new reduction technique which is based on traditional dynamic partial order reduction, but adapted to explore non-sequentially consistent Shasha-Snir traces. The techniques of Papers III and IV are described in some additional detail in respectively Section 4.2 and Section 4.3.

Related Work.
The related work is discussed in the Related Works sections in the papers: Section 8 of Paper I, Section 8 of Paper III, and Section 5 of Paper IV.

Acknowledgments
This work has been partly supported by the Swedish Research Council, partly within the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.
In this chapter we will provide formalisms which are necessary in the rest of the thesis. In Section 2.1 we introduce some basic notation. In Section 2.2 we first introduce a simple assembly-like programming language for concurrent programming with shared memory. Then we formalize its semantics under the common concurrency model sequential consistency [31].

2.1 Preliminaries

Functions.
For a function \( f \), let \( f[x \leftarrow v] \) denote the function \( f' \) such that \( f'(x) = v \) and \( f'(y) = f(y) \) for all \( y \neq x \). We will use lambda expressions to denote anonymous functions. Let \( \lambda x.e \) denote the function \( f \) such that \( f(y) \) evaluates to the value of the expression \( e \) where all free occurrences of \( x \) have been replaced by the value \( y \).

Relations.
We will frequently use various binary relations throughout this thesis. We treat binary relations as sets of pairs, and use the following operators on binary relations. If \( Q \) and \( R \) are binary relations, then \( R^* \) denotes the transitive reflexive closure of \( R \), and \( R^+ \) denotes the transitive irreflexive closure of \( R \). Furthermore \( R^? \) is defined as the union of \( R \) and the identity relation. The composition \( Q; R \) is the relation \( \{(a, c) | \exists b. (a, b) \in Q \land (b, c) \in R\} \).

Words.
A word is a finite sequence of symbols from some alphabet. We let \( \varepsilon \) denote the empty word. For a symbol \( a \), we overload \( a \) to also denote the word consisting of precisely one symbol \( a \). For two words \( w \) and \( w' \), we let \( w.w' \) denote the concatenation of \( w \) and \( w' \).
2.2 Program Model

In this thesis, we will investigate concurrent programs, and how to verify and test them. In this section, we will define the model for concurrent programs, which will be used in the remainder of the text. This entails defining what a concurrent program is, and give an intuition for how it behaves.

2.2.1 Language

First, we will define a programming language for concurrent programs. The language will be small and simple, to benefit presentation. We will also keep the language assembly-like, with very little abstraction. This is because we are here concerned with the effects of relaxed memory. It is necessary for the language to clearly express the precise memory accesses that occur, the order in which they occur, and their various inter-relationships. Therefore, each memory load or store is explicit, as well as the use of thread-local registers.

What is it that is modelled then by a program code expressed in our language? We should think of such a model as corresponding directly to the compiled machine code of an actual program. The model should be interpreted as mirroring the control flow and memory accesses of the machine code. On the other hand, the model does not correspond directly to the source code of the actual program. This is because the source code of the actual program usually does not determine the control flow, memory accesses or register use of the compiled binary. All these things may be determined by the compiler, and will depend on e.g., optimization level and compiler version.

Figure 2.1 presents the grammar of our programming language. Figure 2.2 gives a small example program. A program code starts by a declaration of named global variables. A declaration has the form $x = i$, and declares the existence of a global variable by name $x$ with initial value $i$. The rest of the code consists of a number of thread declarations. Each thread is assigned a unique thread identifier, and a list of uniquely labelled instructions.

An instruction here can be thought of as corresponding to a single machine code instruction. Below, we will first introduce various components occurring in instructions, and then give a brief introduction of each kind of instruction.

An expression, $\langle expr \rangle$, is an arithmetic expression over integer literals and registers. We leave the precise types of allowed operations ($f$) undefined, but note that they will include simple arithmetic like $a_0 + a_1$ for arithmetic expressions $a_0$ and $a_1$, as well as comparisons like $a_0 = a_1$. The latter expression, $a_0 = a_1$, is assumed to evaluate to 1 if $a_0$ and $a_1$ evaluate to the same value, and 0 otherwise. An expression of the form $\& x$ is interpreted as the literal integer which is the address in memory of the global variable $x$. Expressions cannot contain memory accesses. So in order to use the value stored at $x$ in an expression, the value would first need to be loaded into a register using a separate load instruction.
\[
\langle \text{prog} \rangle \ ::= \langle \text{vardecl} \rangle^* \langle \text{thread} \rangle^+
\]
\[
\langle \text{thread} \rangle \ ::= \text{'}\text{thread}'\langle \text{tid} \rangle\text{'}: \langle \text{instr} \rangle^+
\]
\[
\langle \text{instr} \rangle \ ::= \text{'}\text{nop}' \quad \text{// Does nothing}
\]
\[
| \langle \text{reg} \rangle \text{'}:= \langle \text{expr} \rangle \quad \text{// Local assignment}
\]
\[
| \langle \text{reg} \rangle \text{'}:= \langle \text{memloc} \rangle \quad \text{// Memory load}
\]
\[
| \langle \text{memloc} \rangle \text{'}:= \langle \text{expr} \rangle \quad \text{// Memory store}
\]
\[
| \text{'}\text{goto}'\langle \text{label} \rangle \quad \text{// Unconditional branch}
\]
\[
| \text{'}\text{if}'\langle \text{expr} \rangle\text{'}\text{goto}'\langle \text{label} \rangle \quad \text{// Conditional branch}
\]
\[
| \text{'}\text{fence}' \mid \text{'}\text{sync}' \quad \text{// Full memory fence}
\]
\[
| \text{'}\text{sfence}' \mid \text{'}\text{lwsync}' \quad \text{// Other memory fences}
\]
\[
| \text{'}\text{isync}' \quad \text{// Other memory fence}
\]
\[
\langle \text{memloc} \rangle \ ::= \text{['}\langle \text{expr} \rangle\text{']} \mid \langle \text{var} \rangle
\]
\[
\langle \text{vardecl} \rangle \ ::= \langle \text{var} \rangle \text{'}=\text{'Z}
\]
\[
\langle \text{label} \rangle \ ::= \text{regex}('^[A-Z][A-Z0-9]*'$)
\]
\[
\langle \text{tid} \rangle \ ::= \text{regex}('^[A-Z][A-Z0-9]*'$)
\]
\[
\langle \text{var} \rangle \ ::= \text{regex}('^[a-z][a-z0-9]*'$)
\]
\[
\langle \text{reg} \rangle \ ::= \text{regex}('^[r0-9]*$')
\]
\[
\langle \text{expr} \rangle \ ::= \langle \text{reg} \rangle \mid \text{'Z} \mid \text{'}\&'\langle \text{var} \rangle \mid f(\langle \text{expr}_0 \rangle, \cdots, \langle \text{expr}_n \rangle)
\]

Figure 2.1. Grammar of the simple assembly-like language that we will use in this thesis to model concurrent programs.
Registers are thread-local integer variables, named $r_0$, $r_1$, etc. The example program in Figure 2.2 uses five registers: $r_0$ and $r_1$ for $P$, and $r_0$, $r_1$ and $r_2$ for the thread $Q$.

In memory access instructions (loads and stores) memory locations are specified on the form $[a]$ where $a$ is an arithmetic expression. This indicates the location in memory which is addressed by the integer evaluation of $a$. As we will see when we discuss in detail the semantics of our programming language, we assume that every integer addresses a valid and unique memory location. For convenience, we also allow specification of memory locations on the form $x$ where $x$ is a global variable. This is syntactic sugar for $[\&x]$.

The following is a list of the different kinds of instructions supported by our assembly language:

**No-op.** A *nop* instruction does nothing.

**Local Assignment.** A local assignment $r := a$ evaluates the expression $a$ and stores the result in the register $r$.

**Memory Load.** A load instruction $r := \[a\]$ evaluates the arithmetic integer expression $a$, interprets the value as a memory address, and loads the value at that memory location into the register $r$.

**Memory Store.** A store instruction $\[a_0\] := a_1$ evaluates the arithmetic integer expression $a_0$, interprets the value as a memory address, and stores the evaluation of the arithmetic expression $a_1$ into that memory location.

**Unconditional Branch.** An instruction *goto* $l$ changes the control flow of the thread, such that execution continues at the instruction labelled $l$, in the code of the same thread.

**Conditional Branch.** An instruction *if* $a$ *goto* $l$ evaluates the arithmetic expression $a$. If the value is 0, the conditional branch does nothing, and execution continues on the next line of code as usual. If the value is non-zero, the control flow of the thread is changed such that execution continues at the instruction labelled $l$, in the code of the same thread.

**Memory Fences.** Memory fences *fence*, *sync*, *sfence*, *lwsync*, and *isync* enforce memory consistency in various ways. Their effects under the different relaxed memory models are described in Sections 3.2 and 3.3. Under Sequential Consistency, all memory fences execute as *nop* instructions.

**Example 2.1.** Figure 2.2 shows a small example program illustrating a typical idiom for synchronization between threads using shared variables. It consists of two threads $P$ and $Q$. Intuitively, the thread $P$ prepares two arguments for a computation (addition) in the variables $x$ and $y$ (lines $L0$ and $L1$), and passes them to the thread $Q$ using the variable $z$ as a flag to signal that the arguments have been assigned (line $L2$). The thread $Q$ spins on $z$ on lines $M0$ and $M1$ until the arguments are ready. Then it performs
\begin{verbatim}
x = 0 y = 0 z = 0

thread P:  
L0: x := 1;  
L1: y := 2;  
L2: z := 1;  
L3: $r0 := z;  
L4: if $r0 = 1 goto L3;  
L5: $r1 := x;

thread Q:  
M0: $r0 := z;  
M1: if $r0 = 0 goto M0;  
M2: $r1 := x;  
M3: $r2 := y;  
M4: x := $r1 + $r2;  
M5: z := 0;

\end{verbatim}

Figure 2.2. An awkward way of computing 1+2.

the computation (the addition), stores the result back in \(x\), and resets \(z\) to signal that the result is available. The thread \(P\) spins on \(z\) until the result is ready, and then reads the result into its local register \(\$r1\) on line \(L5\).

2.2.2 Program Objects

Given a program code according to the grammar in Figure 2.1 we construct a \textit{program} object \(P\) as follows. A program \(P\) is a tuple on the form \(P = (M^0, T, \lambda^0, \lambda^{\text{next}}, \lambda^{\text{instr}})\). Here \(M^0 : \mathbb{Z} \mapsto \mathbb{Z}\) is a function which for every memory address \(\alpha\) gives its initial value \(M^0(\alpha)\). For each named global variable \(x\) declared in the program code, some unique address \(\alpha_x\) has been chosen, and \(M^0(\alpha_x) = v\) where \(v\) is the initial value given in the declaration. For all other memory addresses \(\alpha\), we have \(M^0(\alpha) = 0\). The set \(T\) contains precisely the thread identifiers declared in the program code. For each thread identifier \(t \in T\), we let \(\lambda^0(t)\) be the label of the first line in the code of the thread identified by \(t\). For each label \(l\) appearing in the program code, we let \(\lambda^{\text{next}}(l)\) be the label of the next line in the code of the same thread. If \(l\) is the last label in the code of its thread \(t\), then we let \(\lambda^{\text{next}}(l) = \bot^t\) where \(\bot^t\) is a special label used to identify the end of the code. For each label \(l\) appearing in the program code, we let \(\lambda^{\text{instr}}(l)\) be the instruction labelled by \(l\) in the program code. We assume that each sub-expression of the form \&\(x\) and memory location of the form \(x\) for some variable \(x\), appearing in any instruction have been replaced by respectively the literal integer \(\alpha_x\) or the memory location \([\alpha_x]\), where \(\alpha_x\) is the chosen address of the variable \(x\).

We let \(\text{Regs}\) denote the set of all possible registers that any thread may use, and \(\text{Labels}\) denote the set of all possible labels including the special labels \(\bot^t\).

\textbf{Example 2.2.} For example, assume that \(P = (M^0, T, \lambda^0, \lambda^{\text{next}}, \lambda^{\text{instr}})\) is the program object of the code given in Figure 2.2. Let us assume that the global variables \(x\), \(y\) and \(z\) are assigned addresses respectively 1, 2 and 3. Then \(M^0\) is the function returning 0 for any integer. The thread identifiers
are $T = \{P, Q\}$, and the initial labels are $\lambda^0(P) = L0$ and $\lambda^0(Q) = M0$. Examples of label successors are $\lambda_{\text{next}}(L0) = L1$ and $\lambda_{\text{next}}(L4) = L5$ and $\lambda_{\text{next}}(L5) = L0$. The addresses of the global variables have been substituted, and so we have e.g., $\lambda_{\text{Instr}}(M2) = (\$r1:=[1])$.

### 2.2.3 Semantics under Sequential Consistency

In the previous section, we gave an intuitive explanation of the instructions of our programming language. Here we will introduce the program semantics under the commonly assumed Sequential Consistency (SC) memory model [31]. We will formalize those semantics, by defining the state transition system $T^SC_P$ induced by a program object $P$.

Under sequential consistency we assume that programs are executed on the following intuitive abstract machine. The machine consists of a number of threads, and one memory, as shown in Figure 2.3. At any time, only one thread may access the memory. The threads each keep track of their current position in their code, and the integer valuation of each of their registers. The memory keeps track of the current value stored at each memory location. The threads take turns, in an arbitrary non-fixed order, to execute instructions according to the control flow of their code. Each instruction is executed atomically, and immediately reads and writes the values in registers and in the memory. Therefore a run is a sequence of instructions executed one by one by the various threads. This sequence is an interleaving of the sequences of instructions executed by each thread.

**Example 2.3 (Run under SC).** Figure 2.4 shows an example of a run of the program of Figure 2.2. On the first line, the thread $P$ executes its first instruction. The memory is immediately updated to store the value 1 at the address associated with the variable $x$. On the next two lines, the memory is similarly updated for $y$ and $z$. On the fourth line, the other thread $Q$ executes a load instruction. It loads the current value (1) in memory at the address associated with $z$ into its local register $\$r0$. On the next line, $Q$
1: P: L0: x := 1 
2: P: L1: y := 2 
3: P: L2: z := 1 
4: Q: M0: $r0 := z  // Reads 1 
5: Q: M1: if $r0 = 0 goto M0  // Do not go to M0 
6: Q: M2: $r1 := x  // Reads 1 
7: P: L3: $r0 := z  // Reads 1 
8: Q: M3: $r2 := y  // Reads 2 
9: Q: M4: x := $r1 + $r2  // Stores 3 into x 
10: Q: M5: z := 0 
11: P: L4: if $r0 = 1 goto L3  // Go to L3 
12: P: L3: $r0 := z  // Reads 0 
13: P: L4: if $r0 = 1 goto L3  // Do not go to L3 
14: P: L5: $r1 := x  // Reads 3 

Figure 2.4. A sample run of the program given in Figure 2.2.

compares the value of $r0 with 0. Since the value is not zero, execution of Q continues at the next label M2 instead of jumping back to M0. On the following three lines, Q loads the variables x and y with values 1 and 2 respectively, and P loads the variable z with the value 1. Notice that the register $r0 used by P is thread-local, and not the same register as the $r0 used by Q. The thread Q continues to compute the sum 3 of the values of its registers $r1 and $r2, and stores that value to x. Thereafter, Q stores the value 0 to z, to signal to P that the computation is done. On line 11, the thread P compares the value 1, previously read from the address of z in memory, with 1. Since the values are equal, the execution of P jumps back to the label L3. On the following two lines, P loads the value of z again and compares it with 1. This time P reads the value 0 (previously written by Q, and execution can continue to the label L5. Finally, P loads the value 3 from memory at the address of x.

Operational Model of Sequential Consistency

Let a program object $P = (M^0, T, \lambda^0, \lambda^{next}, \lambda^{instr})$ be given. In this section, we will define the state transition system $T^{SC}_P$ induced by $P$. The transition system is a directed, labelled graph $T^{SC}_P = (S^{SC}_P, \Delta^{SC}_P)$.

States.
A state $\sigma \in S^{SC}_P$ is a tuple $\sigma = (M, Q, R)$. Here $M : Z \mapsto Z$ is a function which for each memory address $\alpha$ gives the value currently stored at that location $M(\alpha)$. For each thread $t \in T$, its local state is given by $Q(t)$ and $R(t)$. Here $Q(t) \in Labels$ is the label of the next instruction to
be executed by $t$, or $\perp^t$ if $t$ has terminated. The current value of each register $r$ of $t$ is given by $\mathcal{R}(t)(r) \in \mathbb{Z}$. The initial state is defined as follows: $\sigma_P^{0,SC} = (\mathcal{M}^0, \lambda^0, \lambda_t.\lambda_r.0)$, where $\mathcal{M}^0$ and $\lambda^0$ are the initial memory and initial labeling function from the program object $P$.

For a thread-wise register assignment $\mathcal{R} : T \leftrightarrow \text{Regs} \leftrightarrow \mathbb{Z}$ and an arithmetic expression $a$, we will use $\mathcal{R}(t)(a)$ to denote the value of $a$ when evaluated under the register assignment $\mathcal{R}(t)$. I.e., $\mathcal{R}(t)(a) = \mathcal{R}(t)(r)$ when $a = r$ for some register $r$, and $\mathcal{R}(t)(a) = z$ when $a = z \in \mathbb{Z}$, and $\mathcal{R}(t)(a) = f(\mathcal{R}(t)(a_0), \ldots, \mathcal{R}(t)(a_n))$ when $a = f(a_0, \ldots, a_n)$ for some operation $f$ and sub-expressions $a_0, \ldots, a_n$.

Transitions.
A transition is a tuple $(\sigma_0, l, \sigma_1) \in \mathcal{S}^{SC} \times \text{Labels} \times \mathcal{S}^{SC}$ with the intuitive meaning that from the state $\sigma_0 = (\mathcal{M}_0, \mathcal{Q}_0, \mathcal{R}_0)$, some thread $t$ may execute its next instruction, labelled $l = \mathcal{Q}_0(t)$, which causes the abstract machine to reach the next state $\sigma_1$. Notice that each label in the program is associated with one unique line of code. Therefore a label on a transition uniquely defines the executing thread and the executed instruction. We will sometimes write $\sigma_0 \xrightarrow{l} \sigma_1$ to denote that $(\sigma_0, l, \sigma_1) \in \Delta_P^{SC}$.

The set of transitions $\Delta_P^{SC}$ is the smallest set such that $(\sigma_0, l, \sigma_1) \in \Delta_P^{SC}$ holds with $\sigma_0 = (\mathcal{M}_0, \mathcal{Q}_0, \mathcal{R}_0)$ and $\sigma_1 = (\mathcal{M}_1, \mathcal{Q}_1, \mathcal{R}_1)$ whenever one of the following requirements holds for some thread $t$ with $\mathcal{Q}_0(t) = l \neq \perp^t$, and $\mathcal{Q}_0' = \mathcal{Q}_0[t \leftarrow \lambda_{\text{next}}(l)]:$

**No-op.** $\lambda_{\text{instr}}(l) = \text{nop}$ and $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{Q}_1 = \mathcal{Q}_0'$ and $\mathcal{R}_1 = \mathcal{R}_0$.

**Local Assignment.** $\lambda_{\text{instr}}(l) = (r := a)$ for some register $r$ and expression $a$, and we have $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{R}_1 = \mathcal{R}_0[t \leftarrow \mathcal{R}_0(t)[r \leftarrow \mathcal{R}_0(t)(a)]]$ and $\mathcal{Q}_1 = \mathcal{Q}_0'$.

**Memory Load.** $\lambda_{\text{instr}}(l) = (r := [a])$ for some register $r$ and expression $a$, and $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{Q}_1 = \mathcal{Q}_0'$ and $\mathcal{R}_1 = \mathcal{R}_0[t \leftarrow \mathcal{R}_0(t)[r \leftarrow \mathcal{M}_0(a)]]$ whenever one of the following requirements holds for some expressions $a_0$, $a_1$, and $\mathcal{Q}_1 = \mathcal{Q}_0'$ and $\mathcal{R}_1 = \mathcal{R}_0$ and $\mathcal{M}_1 = \mathcal{M}_0[\alpha \leftarrow v]$ where $\alpha = \mathcal{R}_0(t)(a_0)$ and $v = \mathcal{R}_0(t)(a_1)$.

**Unconditional Branch.** $\lambda_{\text{instr}}(l) = (\text{goto } l')$ for some label $l'$, and we have $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{Q}_1 = \mathcal{Q}_0[t \leftarrow l']$ and $\mathcal{R}_1 = \mathcal{R}_0$.

**Conditional Branch (True).** $\lambda_{\text{instr}}(l) = (\text{if } a \text{ goto } l')$ for some expression $a$ and label $l'$, and $\mathcal{R}_0(t)(a) \neq 0$ and $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{R}_1 = \mathcal{R}_0$ and $\mathcal{Q}_1 = \mathcal{Q}_0[t \leftarrow l']$.

**Conditional Branch (False).** $\lambda_{\text{instr}}(l) = (\text{if } a \text{ goto } l')$ for some expression $a$ and label $l'$, and $\mathcal{R}_0(t)(a) = 0$ and $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{R}_1 = \mathcal{R}_0$ and $\mathcal{Q}_1 = \mathcal{Q}_0'$.

**Memory Fences.** $\lambda_{\text{instr}}(l) \in \{\text{fence, sync, sfence, lwsync, isync}\}$ and $\mathcal{M}_1 = \mathcal{M}_0$ and $\mathcal{Q}_1 = \mathcal{Q}_0'$ and $\mathcal{R}_1 = \mathcal{R}_0$. 

18
We can now formalize the concept of a run under Sequential Consistency\(^1\), which we have hitherto been using informally. A run from a state \(\sigma_0\) is a word 
\[ \tau = l_0.l_1.\cdots.l_n \in \text{Labels}^* , \]
such that there are states \(\sigma_1, \cdots, \sigma_{n+1}\) such that 
\[ \sigma_0 \xrightarrow{l_0} \sigma_1 \xrightarrow{l_1} \cdots \xrightarrow{l_n} \sigma_{n+1}. \]
If it is the case that \(\sigma_0 = \sigma^{0,\text{SC}}\), then we will say that \(\tau\) is a run, otherwise we say that \(\tau\) is a run from \(\sigma_0\).

Notice that a given state \(\sigma\) and label \(l\) uniquely determines one state \(\sigma'\) such that \(\sigma \xrightarrow{l} \sigma'\). Therefore, we can define the following notation for the state reached when executing a run. For a run \(\tau = l_0.l_1.\cdots.l_n\) from a state \(\sigma_0\), let \(\tau(\sigma_0)\) denote the state \(\sigma_{n+1}\) such that 
\[ \sigma_0 \xrightarrow{l_0} \sigma_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma_{n+1}, \]
for some intermediate states \(\sigma_1 \cdots \sigma_n\).

---

\(^1\)We will define runs similarly for different memory models. When speak of a run, it should be clear from the context under which memory model it is a run.
In the previous chapter, we considered programs running under the Sequential Consistency (SC) memory model [31]. However, as we will see in this chapter, most contemporary systems do not provide sequential consistency for its software. Instead, they allow memory accesses in the software to appear out of order. These ordering relaxations cause difficulties in software where multiple threads communicate by writing to and reading from shared memory. When the programmer cannot rely on the order in which the various memory stores are observed by other threads, it becomes more difficult to design communication patterns for multiple threads. These difficulties often lead to programmers making mistakes, and introducing errors in the code. To make matters worse, the memory access ordering relaxations often appear very rarely and erratically. This makes it very difficult to detect the errors, and often impossible to reliably reproduce them in a test suite. In this chapter we will first explore the reasons for systems to introduce these relaxed semantics, the general principles behind such semantics, and the kinds of behaviors that they may cause. Then we continue to examine in detail some of the common relaxed memory models which appear in today’s systems.1

3.1 Relaxed Memory: The Why and How

3.1.1 Memory Access Reordering

Hitherto, we have been quite vague when describing what a memory model is. We have mentioned “memory access reorderings,” but not given any real understanding of what that means. The following example will give a first understanding of what a memory access reordering is, and how it can be observed by the programmer.

1A note on terminology: In some contexts the term “relaxed memory” refers to a smaller class of memory consistency models. In this thesis the term will be used for all non-SC memory consistency models.
Example 3.1 (The Dekker Idiom). In Figure 3.1 we see a small program which contains two threads \( P \) and \( Q \), accessing two shared variables \( x \) and \( y \). The thread \( P \) stores to \( x \) and loads from \( y \). The thread \( Q \) symmetrically stores to \( y \) and loads from \( x \). Under sequentially consistent semantics, as described in Section 2.2.3, there are three possible final states after executing this program. \( Q \) could execute entirely after \( P \), in which case we would end up in a final state where \( P \) has read the initial value 0 of \( y \) into its register \( \$r0 \), and \( Q \) has read the value 1, written by \( P \), from \( x \) into its register \( \$r1 \). Symmetrically, we could end up in a different final state, where \( \$r0 \) holds the value 1, and \( \$r1 \) holds the value 0. Another possibility is that both stores \( L0 \) and \( M0 \) are executed in some order, before the loads \( L1 \) and \( M1 \). In this case, we would end up in a final state where both \( \$r0 \) and \( \$r1 \) hold the value 1.

It is impossible, under sequential consistency, to end up in a state where both \( \$r0 \) and \( \$r1 \) hold the value 0. This is because, if \( L1 \) is executed before the store \( M0 \), then since \( L0 \) comes before \( L1 \), and \( M1 \) comes after \( M0 \), the load \( M1 \) must necessarily execute after the store \( L0 \), and will therefore read the value 1.

However, on most current hardware, it is possible to observe final states of the Dekker idiom, where both threads read the value 0. This can be understood as a reordering of memory accesses. In the program code, the store \( L0 \) precedes the load \( L1 \), but if they are allowed to execute in the reverse order, as shown in Figure 3.2, then it is possible to reach a final state where both \( \$r0 \) and \( \$r1 \) hold the value 0. Depending on the hardware and compiler, such a memory access reordering may be possible, and can in this way lead to observable behaviors that would otherwise be impossible.

To see why this phenomenon matters, let us consider the Dekker idiom in a context. The Dekker idiom is in fact a pattern that makes up the core of Dekker’s mutual exclusion protocol [20]. The full algorithm is shown in Figure 3.3. The algorithm depends on the fact that at most one thread can read the initial value zero, and uses this fact to ensure that at most one thread, at a time, can access a critical section. Therefore, in order for the algorithm to be able to guarantee mutual exclusion when running on a non-sequentially consistent system, additional instructions must be added.

\[
\begin{align*}
x &= 0 \\
y &= 0
\end{align*}
\]

thread \( P \): thread \( Q \):
\[
\begin{align*}
L0: & \ x := 1; \\
M0: & \ y := 1; \\
L1: & \ $r0 := y; \\
M1: & \ $r1 := x;
\end{align*}
\]

Figure 3.1. The idiom known as Dekker, or SB.
in order to enforce the necessary memory access ordering (as described later in this section).

The reader may rightly wonder why these strange behaviors appear when we execute a parallel program on a modern system. The reason is performance. If the hardware is not forced to rigidly maintain the observed order between memory accesses, then it is free to implement a wide range of optimizations to improve execution speed as well as energy consumption. For example, the hardware, then, does not need to wait for time consuming earlier memory accesses to complete before starting to execute subsequent instructions. Multiple memory accesses can be sent to the memory system as a batch. The memory may be parallelized with multiple independent memory banks that do not need synchronize with or wait for each other. The processor may speculatively fetch data from the memory early, or even speculate on the value in memory before that value is actually read. Different processor cores on the same chip may share data with each other, without having to synchronize with other, distant, cores on another chip.

Relaxed memory does not reside only in hardware. As well as hardware optimizations may introduce memory access reorderings, so too may compiler optimizations. Many common compiler optimizations, that were originally invented with single-threaded execution in mind, cause observable memory access reorderings when applied to code that runs in parallel. Ubiquitous examples of such optimizations are common subexpression elimination, and any optimization involving code-motion. In most of the remainder of this thesis we will focus on hardware relaxed memory. In this section however, we will speak about memory relaxations in general, and will use the term “system” to denote the entire environment in which a program is compiled and executed. This includes notably both hardware and compiler.

To enable reasoning about which memory access reorderings may appear on a system, the systems provide memory models (also known as consistency models, or memory consistency models). They abstract from the details of the implementation of the system – hardware and compiler optimizations in particular – and instead provide a specification of which memory orderings are guaranteed to be maintained for any software running on the system. Soft-
global \( x = 0, y = 0, t = 0; \)

\[ P() \]

// Acquire lock
1: \( x := 1; \)
2: \( \text{while} (y = 1) \{ \)
3: \( \text{if} (t \neq 0) \{ \)
4: \( x := 0; \)
5: \( \text{while} (t \neq 0) \{ \)
   // Spin
6: \} \)
7: \( x := 1; \)
8: \} \)
9: \} \)

// Crit. sect.
10: ... \)
11: \( t := 1; \)
12: \( x := 0; \)

\[ Q() \]

// Acquire lock
13: \( y := 1; \)
14: \( \text{while} (x = 1) \{ \)
15: \( \text{if} (t \neq 1) \{ \)
16: \( y := 0; \)
17: \( \text{while} (t \neq 1) \{ \)
   // Spin
18: \} \)
19: \( y := 1; \)
20: \} \)
21: \} \)

// Crit. sect.
22: ... \)
23: \( t := 0; \)
24: \( y := 0; \)

\textit{Figure 3.3.} Dekker’s mutual exclusion algorithm. The store on line 1 (alternatively the one on line 7) corresponds to \textbf{L0} in Figure 3.1. The load in the while condition on line 2 corresponds to \textbf{L1}. The thread \( Q \) is symmetrical. For readability, this algorithm is given in an intuitive high level pseudo-code instead of the assembly used in the smaller examples.
ware developers who develop their code to be correct under the given memory model are then guaranteed that their software should run correctly on the system. For hardware, the memory model is usually described in its architecture manual [51, 27, 15, 26]. However, those descriptions are often informal and imprecise, leading to confusion regarding the actual semantics of the architecture (see e.g., [38]). Therefore, significant effort in academia has been spent to determine, and formalize the memory models of common architectures [48, 46, 14, 11]. For memory models induced by compilers, some programming languages, notably Java [36] and C++ [28], now include rigorous, but informal, descriptions of the memory model in their language specification.

### 3.1.2 Shasha-Snir Traces

In their classic work [49], Shasha and Snir introduce a way to represent program runs under relaxed memory as directed graphs. These graphs are known as “Shasha-Snir traces”. They provide a helpful way to identify and illustrate observable behaviors under relaxed memory.

The nodes of the graphs are instances of executed instructions (so called “events”). The edges of the graph are given by four relations over events: the program order, the coherence order, the read-from relation, and the from-read relation. All events executed by the same thread are totally ordered by program order (po) – the order in which they appear in the control flow of the executing thread. For each memory location $x$, all stores to $x$ by any thread are totally ordered by coherence order (co) – the order in which they may be observed by readers. The read-from relation (rf) consists of the pairs $(w, r)$ where $w$ is a store, and $r$ is a load from the same memory location, which reads the value written by $w$. The from-read relation (fr) is given by the coherence order and the read-from relation. It consists of all pairs $(r, w)$ where $(w', r)$ is in the read-from relation, and $(w', w)$ is in the coherence order. I.e., a load is from-read-related to a store $w$ when it reads from a store $w'$ which is earlier than $w$.

A formal definition of Shasha-Snir traces is given in Section 3.3.2.

In [49], Shasha and Snir show the important result that an observable behavior is non-SC iff its Shasha-Snir trace contains a cycle.
Example 3.2 (Shasha-Snir Trace of the Dekker Idiom). In Example 3.1, we showed how the Dekker idiom might be executed such that both threads $P$ and $Q$ read the initial values for variables $y$ and $x$ respectively. Figure 3.4 shows the Shasha-Snir trace corresponding to that run. The load $L_1$ reads the initial value of $y$. The initial value is assumed to be written by an implicit initializer store before the run starts. The initializer store is coherence-before the store $M_0$. Therefore there is a from-read edge from $L_1$ to $M_0$. Similarly, the load $M_1$ is from-read-related to the store $L_0$. Notice that the Shasha-Snir trace has a cycle, showing that its behavior is not possible under SC.

3.1.3 Optimizations and Consistency Relaxation

In this section, we will investigate a number of optimizations that may be performed by a system while running a program, and the effects on memory consistency of those optimizations. The purpose of the section is to familiarize the reader with the kinds of behaviors that may appear under relaxed memory, and understand the underlying technology which causes them. The list of optimizations and behaviors is in no way complete. It is based on knowledge which is common in the area, and described in e.g., [9, 50].

Store Buffering.

A behavior such as the one in Figure 3.4 can be caused by store buffering in a processor core. The idea is the following. Completing a memory store takes a significant amount of time. Depending on the system, it is often necessary to communicate with a distant Last Level Cache (LLC) controller, and even with other processor cores, in order to get exclusive permission to the relevant cache line. This needs to be done before the value can be stored to the cache system, and eventually to the memory. Therefore, if all subsequent instructions need to be stalled while the earlier store completes, there will be a significant performance penalty. The store buffering approach is to keep stores in a local FIFO buffer from the point when the store is committed in the processor core to the point when it is stored to the cache system. Other, subsequent instructions are allowed to execute in the meanwhile, rather than being stalled.

Consider the run shown in Figure 3.2. The reordering of $L_0$ and $L_1$ can be explained by store buffering. The core running $P$ first starts to execute the store $L_0$, and inserts $L_0$ into its store buffer waiting to be able to complete the store. Before $L_0$ can complete, the core executes the load $L_1$, and reads the current value 0 of $y$. Then the core of $Q$ executes and completes both of its instructions, storing 1 to $y$ and loading the value 0 of $x$ from memory. Eventually the core of $P$ gets exclusive permissions for $x$, and finishes storing the value 1.
\[ x = 0 \quad y = 1 \]

\textbf{thread P:} \hspace{1cm} \textbf{thread Q:}
\begin{align*}
L0 & : x := 1; \quad M0: \$r0 := x; \\
L1 & : y := 2; \quad M1: \$r1 := y; \\
L2 & : x := 2;
\end{align*}

\textit{Figure 3.5.} A variation of the MP idiom, where \( W \rightarrow W \) reordering may appear as a result of store coalescing.

\[ L0 : x := 1 \]
\begin{tikzpicture}
  \node [operation] {\text{po}};
  \node [load] (L1) at (1,0) {L1: y := 2};
  \node [store] (M0) at (2,0) {M0: \$r0 := x};
  \node [load] (M1) at (3,0) {M1: \$r1 := y};
  \node [store] (L2) at (4,0) {L2: x := 2};
  \draw [->] (L1) -- (M0);
  \draw [->] (M0) -- (M1);
  \draw [->] (M1) -- (L2);
  \draw [->] (L2) -- (L1);
\end{tikzpicture}

\textit{Figure 3.6.} A Shasha-Snir trace showing the observable effect of store coalescing in the program in Figure 3.5.

Store buffering is observable as so called \( W \rightarrow R \) reordering where loads are allowed to overtake earlier stores by the same core. It is the main motivation for the TSO memory model, described in Section 3.2.

\textit{Store Coalescing.}
A natural extension of the store buffer technique is the use of store coalescing. If a store \( x := 1 \) is pending in the store buffer and another store \( x := 2 \) is committed by the same processor core, then instead of having both stores in the store buffer, the first store of the value 1 can be replaced in the store buffer by the store of the value 2, and the second store can then complete early.

Store coalescing gives rise to additional observable memory consistency relaxations, on top of those introduced by store buffering. It allows \( W \rightarrow W \) reordering, where stores by the same core are reordered. Consider the program in Figure 3.5. On an SC system, or a system employing only store buffering as described above, any final state of that program will satisfy the condition that \( \$r0 \leq \$r1 \) in the thread \( Q \). This is because in a run where \( M0 \) reads the value 2 written by \( L2 \) to \( x \), the load \( M1 \) must necessarily occur after the store \( L1 \) completes. With store coalescing, this is no longer the case. In a run where \( P \) first commits all stores \( L0, L1 \) and \( L2 \) in order, but then coalesces \( L2 \) with \( L0 \), it is possible for the value of \( L2 \) to become available in memory before the value of \( L1 \). Then it is possible for the other thread \( Q \), to read first the value 2 from \( x \) and then the initial value 1 from \( y \). Figure 3.6 shows the corresponding Shasha-Snir trace.
\[ x = 0 \quad y = 0 \]

\begin{align*}
\text{thread P:} & \quad \text{thread Q:} \\
L0: & \quad x := 1; \quad \text{M0:} \quad r0 := y; \\
L1: & \quad y := 1; \quad \text{M1:} \quad \text{if} r0 = 0 \text{ goto M3;}
\quad \text{M2:} \quad r1 := x; \\
\text{M3:} & \quad \text{nop;}
\end{align*}

Figure 3.7. The MP+ctrl idiom. There is a control dependency from M0 to M2.

Parallel Cache Banks.
Since many cores may simultaneously try to access memory, the single LLC may become a bottleneck. One approach to alleviate the problem is to separate the LLC into multiple, parallel cache banks, each responsible for a separate part of the memory.

Having multiple cache banks enables reorderings of all four basic kinds: \( R \to R \) and \( R \to W \) and \( W \to R \) and \( W \to W \). This is because different banks may respond to requests with different latency, e.g., when one of them is under a heavier load than the others. Then some memory accesses may finish quickly, while others get delayed.

Speculative Loads.
As we are starting to see, it is often possible to let multiple instructions, and memory accesses in particular, execute simultaneously in order to hide their latencies. Conditional branches present an obstacle to this approach. If a branch condition depends on a pending load instruction, the core cannot yet know which instructions should be executed after the branch. Speculation is a technique that alleviates this problem, by allowing the core to guess whether or not the branch will be taken. Instructions along the guessed control flow may then start to execute, as long as they do not perform any operations that are irrevocable or visible outside the core. If the guess should turn out to be false, then the results of the speculatively executed instructions are discarded. In particular, systems with relaxed memory models may permit speculatively executed loads to read values from memory before it is determined that the loads are on the correct control flow path.

When an instruction \( i \) lies on a control flow path which is dependent on a conditional branch which depends on the value read by an earlier load event \( i_r \), we say that there is a control dependency from \( i_r \) to \( i \). The optimization of speculative loads may cause observable \( R \to R \) reorderings even between loads where the latter is control dependent on the former. Consider the program in Figure 3.7. It has a control dependency between M0 and M2. However, with load speculation, the system is nonetheless allowed to satisfy the load M2...
Figure 3.8. A non-SC execution of the program in Figure 3.7. In the presence of load speculation this behavior is observable, even if the order between the stores L0 and L1 is enforced.

```
global x = 0, y = 0;
Q()
1: local _a := x*x;
2: ...
3: local _b := y;
4: ...
5: local _c := x*x;
```

```
thread Q:
M0: $r0 := x;
M1: $r0 := $r0*$r0;
M2: $r1 := $r0; // a
M3: $r2 := y; // b
M4: $r3 := $r0; // c
```

Figure 3.9. Left: Pseudo code where lines 1 and 5 share a common subexpression. Right: Compiled assembly version of the program to the left, where common subexpression elimination has been performed.

before the load M0. This may lead to an observable non-SC behavior such as the one shown in Figure 3.8.

**Common Subexpression Elimination.**

One very common optimization performed by compilers is common subexpression elimination. Consider the pseudo code in Figure 3.9 (left). When compiling such code, the optimizing compiler may recognize that the lines 1 and 5 share a common subexpression \(x^2\). Rather than letting this expression be computed twice, the compiler may optimize by computing the expression only once, store it in a register, and use the same value at both places in the code. The code in Figure 3.9 (left) could then be compiled into the optimized assembly shown in Figure 3.9 (right).

This common optimization gives rise to \(R \rightarrow R\) memory access reorderings, as the accesses to \(x\) on line 5 in the original code are effectively moved to line 1. Figure 3.10 shows a (pseudo code based) Shasha-Snir trace displaying a non-SC behavior that may appear as a result. What happens in the run that gives rise to the trace is that the thread on the right first executes the assembly lines M0-M2, and reads the initial value 0 for \(x\). Then the other thread performs both
\( x := 1 \)
\( y := 1 \)
\( \text{local } _a := x^2 \)
\( \text{local } _b := y \)
\( \text{local } _c := x^2 \)

**Figure 3.10.** A Shasha-Snir trace illustrating an observable non-SC behavior that may be caused by the common subexpression elimination described in Figure 3.9. Here it is assumed that the compiled assembly of the thread on the right is that shown in Figure 3.9 (right).

\[ \begin{align*}
K0: \ x &:= 1 \\
L0: \ y &:= 1 \\
M0: \ r0 &:= x \\
M1: \ r1 &:= y \\
N0: \ r2 &:= y \\
N1: \ r3 &:= x
\end{align*} \]

**Figure 3.11.** IRIW: A behavior caused by non-atomic writes. Here the threads \( R \) and \( S \) observe the stores \( K0 \) and \( L0 \) in different orders.

of its stores. The thread on the right then executes its remaining lines \( M3 \) and \( M4 \). Thus the thread on the right observes the value 1 for \( y \), but the value put in the local variable \( _c \) (corresponding to register \( r3 \) ) is the original value 0, that was loaded from \( x \) at the beginning of the run.

**Cache Sharing.**
On some systems, it may incur more latency for a core to communicate with some cores, than with others. This can happen for example in multi-processor systems, where each processor has multiple cores. The cores on the same processor may communicate efficiently through a shared cache on the same chip, while it takes longer to communicate with cores on a different processor.

This may cause a phenomenon known as **non-atomic stores**, where not only are memory stores reordered, but the different threads cannot even agree on which order they ended up in after the fact. Consider the Shasha-Snir trace in Figure 3.11. Let us assume that the loads in threads \( R \) and \( S \) are executed in order, and not reordered by any of the techniques described above. Here the
thread \( R \) reads the value of \( x \) that was written by \( P \), but then reads the initial value of \( y \) rather than the value that was written by \( Q \). Hence from the perspective of \( R \), the store \( K_0 \) takes place earlier than \( L_0 \). The thread \( S \) is symmetric, and from its perspective the order between \( K_0 \) and \( L_0 \) is the opposite. We could explain this behavior by assuming that the threads \( P \) and \( R \) reside on the same chip and communicate efficiently through a shared cache, while \( Q \) and \( S \) reside on another chip and communicate efficiently with each other but not with \( P \) and \( R \). That would explain why the store \( L_0 \) arrives late to \( R \), and \( K_0 \) is late to \( S \).

Non-atomic stores feature in particular in the POWER [26] and ARM [15] hardware architectures.

3.1.4 Restoring Order

Above we have explored many kinds of relaxed consistency behaviors and the underlying techniques which give rise to them. In this section we will investigate a few techniques, formalizations and notions that allow a developer to regain sufficient order in their software for it to function correctly.

Memory Model Formalization.

In order for developers to be able to reason about the correctness of their programs in the presence of relaxed memory, the semantics of programs running under the memory model must be formalized. Formal semantics are also necessary for formal verification, as well as for systematic testing.

Architecture manuals often contain informal natural language descriptions of the architectures’ memory consistency models [51, 27, 15, 26]. Significant effort has been spent by various academic groups to invent formalizations of those models, and testing that the formalizations are accurate [48, 46, 14, 11]. Usually the goal is for the formalization to be as strong as possible, while being no stronger than (i) the behaviors that have been observed in actual hardware, and (ii) the perceived intent of the architects.

There are two main approaches used for formal memory model semantics: operational semantics [48, 46] and axiomatic semantics [14, 11].

An operational semantics is an abstract machine, which given a program may perform various transitions to change its state. Our SC semantics given in Section 2.2.3 is an example of an operational semantics. While operational semantics can give some intuition for the reason why certain behaviors may appear in hardware, there is not necessarily any connection between the construction of the abstract machine and the actual machine that it models.

Axiomatic semantics focus on describing which observable behaviors are allowed (possible) or forbidden (impossible) in the memory model, without providing any explanation for how those behaviors come to appear in the actual machine. Formally, an axiomatic semantics is a predicate over Shasha-Snir
traces. The predicate is satisfied by precisely those Shasha-Snir traces that correspond to allowed behaviors. Such predicates are usually formalized by defining a number of additional relations over the various events in a Shasha-Snir trace, and a number of requirements, such as acyclicity or irreflexivity, on those relations.

Axiomatic semantics are concerned with the Shasha-Snir traces of complete runs of a program, while operational semantics reason about a chain of transitions from initial to final state.

In Section 2.2.3, we gave an operational semantics for SC. In Section 3.2 we will give the standard operational semantics for the TSO and PSO memory models. In Section 3.3 we will recall the axiomatic semantics of POWER formalized by Alglave et al. in [14].

Fences.
In parallel code, it is common that multiple threads communicate by stores and loads to shared memory. This communication follows some programmer-defined protocol, which often depends on the precise order of certain memory accesses. For example, in Dekker’s mutual exclusion algorithm (Figure 3.3) we have already seen that the correct working of the algorithm depends on the store on line 1 happening before the load on line 2.

In such cases, the programmer needs to figure out which memory access orderings are required for their algorithm to work. In order to allow such necessary orderings to be enforced, the system (hardware architecture or programming language) provides special fence instructions, which can be inserted by the programmer as necessary.

Different memory models may provide fence instructions with different semantics. All of them have in common that they restrict memory access reordering, and so restricts the set of allowed Shasha-Snir traces for the program. Often one can think of the fences as forcing some pairs of memory accesses by the same thread to take effect in an order that coincides with program order.

For example the PSO memory model provides a fence sfence which has the effect that all stores that appear before the sfence in program order in the same thread must take effect before all stores that appear after the sfence in program order in the same thread. The TSO memory model provides only one fence fence. It is a full memory fence: It has the effect that all memory accesses (load or store) that appear before the fence in program order in the same thread must take effect before all memory accesses that appear after the fence in program order in the same thread. These intuitions will be formalized in Section 3.2.

Example 3.3 (Fencing Dekker for TSO). In order to make Dekker’s algorithm work under the TSO memory model, it is necessary to enforce the order between the stores on lines 1 and 7 and the load on line 2. We can do
global x = 0, y = 0, t = 0;

\textbf{P()} \\
// Acquire lock
1: x := 1;
2: \textbf{fence};
3: \textbf{while} (y = 1)\{
4: \quad \textbf{if} (t \neq 0)\{
5: \quad \quad x := 0;
6: \quad \quad \textbf{while} (t \neq 0)\{
\quad \quad \quad \text{// Spin}
7: \quad \quad \}
8: \quad x := 1;
9: \quad \textbf{fence};
10: \}
11: \} \\
// Crit. sect.
12: ... \\
// Release lock
13: t := 1;
14: x := 0;

\textbf{Q()} \\
// Acquire lock
15: y := 1;
16: \textbf{fence};
17: \textbf{while} (x = 1)\{
18: \quad \textbf{if} (t \neq 1)\{
19: \quad \quad y := 0;
20: \quad \quad \textbf{while} (t \neq 1)\{
\quad \quad \quad \text{// Spin}
21: \quad \quad \}
22: \quad y := 1;
23: \quad \textbf{fence};
24: \}
25: \} \\
// Crit. sect.
26: ...
// Release lock
27: t := 0;
28: y := 0;

\textit{Figure 3.12.} Dekker's mutual exclusion algorithm, with fences inserted for correct operation under the TSO memory model.
this by inserting two fence instructions per thread, between each of those pairs, as shown in Figure 3.12.

Coherence.

Some desirable ordering properties are guaranteed by most memory models. One such property is coherence. Intuitively, coherence boils down to a guarantee that for any run under the memory model, if one restricts one’s view to only one memory location, then the run behaves as if sequentially consistent.

More formally, coherence guarantees that for any run under the memory model, and any memory location $x$, there exists a total order $<$\text{co}$_x$ over all memory accesses in the run by any thread to $x$ such that (i) for any two memory accesses $a$ and $b$ to $x$, performed by the same thread where $a$ precedes $b$ in program order, we also have $a <$\text{co}$_x b$, and (ii) any load $b$ from $x$ reads the value written by the $<$\text{co}$_x$-greatest store $a$ to $x$ for which $a <$\text{co}$_x b$.

**Example 3.4** (Incoherent Executions). Figure 3.13 illustrates the definition of coherence, by showing two Shasha-Snir traces which both violate coherence.

In subfigure (a), the load $L2$ reads an old value of $x$, despite the fact that the same thread overwrote that value in $L1$. There is no total order $<$\text{co}$_x$ which is a witness for coherence, since any total order over $L0$, $L1$ and $L2$ must either violate condition (i) or (ii), when ordering $L1$ with $L2$.

In subfigure (b), the loads $M0$ and $M1$ read the values written by the other thread in an order which contradicts the program order between $L0$ and $L1$.

**Example 3.5** (Coherence Does not Imply Consistency). Consider any of the figures 3.4, 3.6, 3.8, 3.10, 3.11 used in previous examples. They all satisfy the coherence property, despite being non-SC. This is because all of their non-SC behaviors involve multiple different memory locations. For example, in Figure 3.4, there are two total orders $<$\text{co}$_x$ and $<$\text{co}$_y$ with $M1 <$\text{co}$_x L0$ and $L1 <$\text{co}$_y M0$. Here, the orders $<$\text{co}$_x$ and $<$\text{co}$_y$ are witnesses to the coherence property. Note, however, that they are only acyclic when
global $x = 0$;

<table>
<thead>
<tr>
<th>$P()$</th>
<th>$Q()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $x := 1$;</td>
<td>2: $x := 2$;</td>
</tr>
</tbody>
</table>

Figure 3.14. A simple program with a data race.

global $x = 0$;
global $L = \text{UNLOCKED}$;

<table>
<thead>
<tr>
<th>$P()$</th>
<th>$Q()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: acquire($L$);</td>
<td>4: acquire($L$);</td>
</tr>
<tr>
<td>2: $x := 1$;</td>
<td>5: $x := 2$;</td>
</tr>
<tr>
<td>3: release($L$);</td>
<td>6: release($L$);</td>
</tr>
</tbody>
</table>

Figure 3.15. The simple program from Figure 3.14, made data race free using mutual exclusion primitives. Here, we assume that the calls to acquire and release are considered to be synchronization operations under the memory model.

considered in isolation. Their union, together with program order contains a cycle, and is not a total order.

The DRF Guarantee.
Another common desirable property that is often maintained in some form by memory models is the “DRF guarantee” [10]. It promises intuitively that if the programmer writes their code in a disciplined manner, such that it contains no data races, then the system guarantees that every run of the program behaves the same as some SC run of the program.

More formally, the system defines some operations which are considered synchronization operations. These can be e.g., atomic read-modify-write instructions, calls to acquire and release functions for a mutual exclusion lock provided by a library, specially identified statements in a programming language, or statements accessing specially marked variables in a programming language. A data race is then defined as two memory accesses by different threads in an SC run, which access the same memory location, and at least one of which is a store, and at least one of which is not a synchronization operation, and which are not separated by any synchronization operation in the run. A program is said to be data race free (DRF) if there is no SC run of the program that contains a race. Figures 3.14, 3.15 and 3.16 illustrate the definition of the DRF property for programs.

The DRF guarantee for a memory model states that if a program is DRF, then for every run $\tau$ of the program under the memory model, there is a run $\tau'$ of the program under SC, such that the Shasha-Snir trace of $\tau$ is the same
global x = 0;
global y = 0;

\textbf{P}()
\begin{align*}
1: & \textbf{if} (x \neq 0) \{ \\
2: & \quad y := 1; \\
3: & \} \\
\end{align*}

\textbf{Q}()
\begin{align*}
4: & \textbf{if} (y \neq 0) \{ \\
5: & \quad x := 1; \\
6: & \} \\
\end{align*}

Figure 3.16. A data-race free program with no locks. Here under SC, none of the then-clauses will execute. Therefore, no race appears in any SC run of the program, and so the program is DRF.

as the Shasha-Snir trace of \( \tau' \). I.e., any behavior that can be observed for the program under the memory model also corresponds to some SC run.

The reason for the popularity of the DRF guarantee is that the definition of the DRF property for programs reasons only about runs under SC. Therefore, a programmer can ensure that the program is correct under a relaxed memory model providing the DRF guarantee, without reasoning about runs under that memory model.

It follows as a corollary from the definition of the DRF guarantee that any single-threaded program will behave as under SC, when executed under a memory model which provides the DRF guarantee. This is because no race can appear in any run of a single-threaded program.

3.2 TSO and PSO

The memory models TSO and PSO are described in the SPARC architecture manual [51], and used by that architecture. In [48], Sewell et al. perform thorough testing of x86 processors and conclude that the memory model of x86 is also essentially TSO. In this section we give formalizations of the standard operational semantics of TSO and PSO. First, we give an intuition of the workings of their abstract machines. Then we formalize the operational semantics.

3.2.1 Semantics of TSO

\textbf{Intuitive Semantics}

The TSO memory model is motivated by store buffering, as described in Section 3.1.3. The only allowed kind of memory access reordering is \( W \rightarrow R \). I.e., loads are allowed to overtake stores to different memory locations.

In the abstract machine of TSO, each thread executes instructions one by one according to the program’s control flow, in the same way as under SC. However, each thread is equipped with a first-in-first-out store buffer. When
a thread executes a store, it does not immediately update the memory with the new value. Instead the store is entered into the thread’s store buffer. Non-deterministically, at any point in time, a store may be dequeued from the store buffer of any thread. The memory is then updated according to the dequeued store. This is called an *update* event.

When a thread executes a load of a memory location $x$, it typically reads the value directly from memory, in the same way as under the SC semantics. However, if the same thread has previously written to $x$, and that store is still pending, then reading from memory would cause the thread to miss the previous store and violate coherence, similar to the scenario shown in Figure 3.13 (a). Therefore TSO employs a technique known as *buffer-forwarding*, or *read-own-write-early* (ROWE): Before reading from memory, a load must check the contents of the store buffer of its own thread. If the store buffer contains a store to the same memory location, the load must read the value from the most recently enqueued such store, instead of reading from memory.

The buffering abstract machine of TSO models how stores may be delayed arbitrarily in store buffers, and thus reordered with subsequent loads by the same thread.

The TSO memory model provides one kind of fence instruction: **fence**. In the abstract machine, when a **fence** is executed, the thread blocks until its own store buffer is empty. This has the effect that a store which is executed before the fence cannot be delayed past any load which is executed after the fence. Hence the fence intuitively maintains the order between any memory access before the fence and any memory access after the fence.

**Example 3.6** (Run of Dekker under TSO). In Figure 3.18, we recall the Dekker idiom that we encountered earlier in this section. Figure 3.19 gives an example run of the idiom under TSO. In step 1, $P$ enqueues the store $L0$ into its store buffer, and then in step 2 the thread reads the value 0 of $y$ from the memory. In steps 3-5, $Q$ enqueues $M0$ to its store buffer, then dequeues
\[ x = 0 \quad y = 0 \]

**thread P:**
\[
\begin{array}{l}
L0: \ x := 1; \\
L1: \ \$r0 := y;
\end{array}
\]

**thread Q:**
\[
\begin{array}{l}
M0: \ y := 1; \\
M1: \ \$r1 := x;
\end{array}
\]

*Figure 3.18.* Recall the Dekker idiom from Figure 3.1.

1. \[P: \ L0: \ x := 1\]
2. \[P: \ L1: \ \$r0 := y\]
3. \[Q: \ M0: \ y := 1\]
4. \[Q: \ \text{update}\]
5. \[Q: \ M1: \ \$r1 := x\]
6. \[P: \ \text{update}\]

*Figure 3.19.* Left: An example run under TSO of the Dekker idiom. Right: The state of store buffers and memory immediately after step 4 of the run to the left.

it and updates the memory with the value 1 for \(y\), and then proceeds to read the value 0 of \(x\) from the memory. Notice that the store \(L0\) of \(P\) is not visible to \(Q\) at this point. Lastly, in step 6, the store \(L0\) updates to memory, and assigns 1 to \(x\).

**Formal Semantics**

Let a program object \(P = (M^0, T, \lambda^0, \lambda_{\text{next}}, \lambda_{\text{instr}})\) be given. In this section, we will define the state transition system \(T_{\text{TSO}}^P\) induced by \(P\). The transition system is a directed, labelled graph \(T_{\text{TSO}}^P = (S_{\text{TSO}}^P, \Delta_{\text{TSO}}^P)\).

**States.**

A state \(\sigma \in S_{\text{TSO}}^P\) is a tuple \(\sigma = (M, Q, R, B)\). The elements \(M, Q\) and \(R\) are interpreted as in the SC semantics: The element \(M : Z \mapsto Z\) is a map from each memory address to the current value held in memory at that address. For each thread \(t \in T\), we have that \(Q(t) \in \text{Labels}\) is the label of the next instruction to execute for \(t\), or \(\bot\) if \(t\) has terminated. For each thread \(t \in T\) and register \(r\), we have that \(R(t)(r) \in Z\) is the current value of that register for \(t\).

The element \(B : T \mapsto (Z \times Z)^*\) represents the current store buffer contents of each thread. For each thread \(t \in T\), we have that \(B(t)\) is a word over pairs \((\alpha, v) \in Z \times Z\), where each pair \((\alpha, v)\) represents a store of the value \(v\) to the memory address \(\alpha\). Stores are enqueued on the right side of the word, and dequeued on the left side.

The initial state is defined as follows: \(\sigma_{\text{TSO}}^0 = (M^0, \lambda^0, \lambda_t: \lambda_r: 0, \lambda_t: \varepsilon)\).

For a store buffer \(B(t)\) and a memory address \(\alpha\), we use the notation \(B(t)(\alpha)\) to denote the value \(v \in Z\) such that \(v = v'\) for the rightmost (i.e., most recent)
pair \((\alpha', v')\) in \(B(t)\) such that \(\alpha' = \alpha\). If there is no such pair in \(B(t)\), then we let \(B(t)(\alpha) = \bot\). Intuitively, \(B(t)(\alpha)\), if any, is the value in the store buffer that would be read by a load from the address \(\alpha\) by \(t\) in case of read-own-write-early.

**Example 3.7** (A TSO State). In Figure 3.19 (right) we saw a graphical representation of a state of the abstract machine executing the Dekker idiom. Formally, this state is represented as \((\mathcal{M}, Q, R, B)\) where the memory is \(\mathcal{M} = \{(\alpha_x, 0), (\alpha_y, 1)\}\), and \(Q = \{ (\mathbf{P}, \bot^2), (\mathbf{Q}, \mathbf{M1}) \}\), and the local register valuation is \(R = \{ (\mathbf{P}, \lambda r.0), (\mathbf{Q}, \lambda r.0) \}\), and \(B = \{ (\mathbf{P}, (\alpha_x, 1)), (\mathbf{Q}, \varepsilon) \}\). Here we assume that \(\alpha_x\) and \(\alpha_y\) are the addresses of \(x\) and \(y\) respectively.

**Transitions.**

A transition is a tuple \((\sigma_0, l, \sigma_1) \in \Delta_P^{TSO}\), where \(\sigma_0, \sigma_1 \in \mathcal{S}_P^{TSO}\) are respectively the source and the target state of the transition, and \(l\) is a description of how the abstract machine may transition from \(\sigma_0\) to \(\sigma_1\). As under SC, \(l\) may be a label, meaning that a thread executes the instruction labeled by \(l\). In addition, under TSO, \(l\) may be a thread identifier, meaning that the thread \(l\) updates the memory with the oldest store from its store buffer. We will sometimes write \(\sigma_0 \xrightarrow{l} \sigma_1\) to denote that \((\sigma_0, l, \sigma_1) \in \Delta_P^{TSO}\).

The set of transitions \(\Delta_P^{TSO}\) is the smallest set such that \((\sigma_0, l, \sigma_1) \in \Delta_P^{TSO}\) holds with \(\sigma_0 = (\mathcal{M}_0, Q_0, R_0, B_0)\) and \(\sigma_1 = (\mathcal{M}_1, Q_1, R_1, B_1)\) whenever one of the following requirements holds for some thread \(t\). Here we let \(Q'_t\) denote the updated program counters \(Q_0[t \leftarrow \lambda^{\text{next}}(l)]\) if \(l \in \text{Labels} \) and \(l \neq \bot^t\), and we let \(Q'_0 = Q_0\) otherwise. For convenience, we use (*), to mark the rules which differ from the corresponding SC semantics in an interesting way.

**No-op.** \(l \in \text{Labels} \) and \(l \neq \bot^t\) and \(\lambda^{\text{instr}}(l) = \text{nop}\) and \(\mathcal{M}_1 = \mathcal{M}_0\) and \(Q_1 = Q'_0\) and \(R_1 = R_0\) and \(B_1 = B_0\).

**Local Assignment.** \(l \in \text{Labels} \) and \(l \neq \bot^t\) and \(\lambda^{\text{instr}}(l) = (r := a)\) for some register \(r\) and expression \(a\), and \(\mathcal{M}_1 = \mathcal{M}_0\) and \(Q_1 = Q'_0\) and \(R_1 = R_0[t \leftarrow R_0(t)[r \leftarrow R_0(a)]]\).

**Read From Memory (*).** \(l \in \text{Labels} \) and \(l \neq \bot^t\) and \(\lambda^{\text{instr}}(l) = (r := [a])\) for some register \(r\) and expression \(a\) and furthermore \(B_0(t)(\alpha) = \bot\), and \(\mathcal{M}_1 = \mathcal{M}_0\) and \(Q_1 = Q'_0\) and \(R_1 = R_0[t \leftarrow R_0(t)[r \leftarrow M_0(\alpha)]]\) where the loaded address is \(\alpha = R_0(t)(a)\) and \(B_1 = B_0\).

**Read-Own-Write-Early (*).** \(l \in \text{Labels} \) and \(l \neq \bot^t\) and \(\lambda^{\text{instr}}(l) = (r := [a])\) for some register \(r\) and expression \(a\) and furthermore \(B_0(t)(\alpha) = v \neq \bot\), and \(\mathcal{M}_1 = \mathcal{M}_0\) and \(Q_1 = Q'_0\) and \(R_1 = R_0[t \leftarrow R_0(t)[r \leftarrow v]]\) where the loaded address is \(\alpha = R_0(t)(a)\) and \(B_1 = B_0\).

**Memory Store (*).** \(l \in \text{Labels} \) and \(l \neq \bot^t\) and \(\lambda^{\text{instr}}(l) = ([a_0] := a_1)\) for some expressions \(a_0, a_1\), and \(Q_1 = Q'_0\) and \(R_1 = R_0\) and \(\mathcal{M}_1 = \ldots\)
\[ M_0 \text{ and } B_1 = B_0[t \leftarrow B_0(t) \cdot (\alpha, v)] \text{ where } \alpha = \mathcal{R}_0(t)(a_0) \text{ and } v = \mathcal{R}_0(t)(a_1). \]

**Unconditional Branch.** \( l \in \text{Labels} \) and \( l \not\perp t \) and \( \lambda^{\text{instr}}(l) = (\text{goto } l') \) for some label \( l' \), and \( M_1 = M_0 \) and \( Q_1 = Q_0[t \leftarrow l'] \) and \( R_1 = R_0 \) and \( B_1 = B_0 \).

**Conditional Branch (True).** \( l \in \text{Labels} \) and \( l \not\perp t \) and for some expression \( a \) and label \( l' \) it holds that \( \lambda^{\text{instr}}(l) = (\text{if } a \text{ goto } l') \) and furthermore \( \mathcal{R}_0(t)(a) \neq 0 \), and \( M_1 = M_0 \) and \( R_1 = R_0 \) and \( Q_1 = Q_0[t \leftarrow l'] \) and \( B_1 = B_0 \).

**Conditional Branch (False).** \( l \in \text{Labels} \) and \( l \not\perp t \) and for some expression \( a \) and label \( l' \) it holds that \( \lambda^{\text{instr}}(l) = (\text{if } a \text{ goto } l') \) and furthermore \( \mathcal{R}_0(t)(a) = 0 \), and \( M_1 = M_0 \) and \( R_1 = R_0 \) and \( Q_1 = Q_0'[t \leftarrow l'] \) and \( B_1 = B_0 \).

**Memory Fence (\(*\)).** \( l \in \text{Labels} \) and \( l \not\perp t \) and \( \lambda^{\text{instr}}(l) = \text{fence} \) and also \( B_0(t) = \varepsilon \), and \( M_1 = M_0 \) and \( Q_1 = Q_0' \) and \( R_1 = R_0 \) and \( B_1 = B_0 \).

**Memory Update (\(*\)).** \( l = t \not\in \text{Labels} \) and \( B_0(t) = (\alpha, v) \cdot \bar{b} \) for some address \( \alpha \), value \( v \) and word \( \bar{b} \), and \( M_1 = M_0[\alpha \leftarrow v] \) and \( Q_1 = Q_0 \) and \( R_1 = R_0 \) and \( B_1 = B_0[t \leftarrow \bar{b}] \).

Based on this semantics, we define the concepts of a run \( \tau \) under TSO and the reached state \( \tau(\sigma) \) analogously to how they are defined for SC.

### 3.2.2 Semantics of PSO

**Intuitive Semantics**

The PSO memory model is mostly the same as TSO. The difference is that instead of having one store buffer per thread, PSO provides one store buffer per thread and memory location. These store buffers update to memory independent of each other, which means that stores to different memory locations may reach memory in an order opposite to the order in which they were inserted into their respective buffers. It follows that under PSO, memory access reorderings of the kind \( W \to W \) are allowed. Figure 3.20 shows the abstract machine of the PSO memory model.

**Example 3.8 (MP under PSO).** Figure 3.21 (left) shows the MP (“Message Passing”) idiom, and a run under PSO of that program (right). On lines 1-2, the stores \( L_0 \) and \( L_1 \) are enqueued in their respective store buffers by \( P \). Figure 3.22 shows the state after these two steps have completed. On line 3, the memory is updated with the value 1 for \( y \), and in the next two steps \( Q \) reads the values 1 and 0 for \( y \) and \( x \) respectively. Notice that under SC and TSO, it would be impossible for \( Q \) to observe those values. Finally, on line 6, the memory is updated with the value 1 for \( x \).
Figure 3.20. The abstract machine of the PSO memory model. Each thread $t_0$, $t_1$, ..., $t_n$ has one store buffer per memory location.

\[
\begin{align*}
    & \text{x = 0 y = 0} \\
    & \text{thread P: thread Q:} \\
    & \text{L0: x := 1; M0: $r0 := y$;} \\
    & \text{L1: y := 1; M1: $r1 := x$;} \\
    & \text{1: P: L0: x := 1} \\
    & \text{2: P: L1: y := 1} \\
    & \text{3: P: update(y)} \\
    & \text{4: Q: M0: $r0 := y$} \\
    & \text{5: Q: M1: $r1 := x$} \\
    & \text{6: P: update(x)}
\end{align*}
\]

Figure 3.21. Left: The MP idiom. Right: A run under PSO of the MP idiom. Notice that the order of dequeueing L0 and L1 from the store buffers of P is the opposite of the order of enqueueing. This allows Q to read the value 1 for y and 0 for x, which would be impossible under both SC and TSO.

Figure 3.22. The state of the abstract machine immediately after step 2 of the run in Figure 3.21.
We consider two different fences under PSO: the full fence `fence`, and the store fence `sfence`. The full fence `fence` works the same under PSO as under TSO. I.e., it intuitively enforces the order between any memory access that precedes the fence and any memory access that follows the fence in program order. The store fence `sfence` is weaker. It too enforces order between memory accesses that surrounds it, but only enforces the W → W order. So an `sfence` enforces the order between any store that precedes it and any store that follows it in program order. But a store and a load which are separated in program order by an `sfence` may still be reordered.

In the abstract machine, the behavior of an `sfence` is modelled as follows: When an `sfence` instruction is executed, it does not pose any immediate requirements on the contents of the store buffers as a full fence does. Instead, the `sfence` is itself enqueued at the end of each store buffer of the executing thread. When an `sfence` reaches the left side of a store buffer, no more stores can be dequeued from that store buffer until the `sfence` has been removed. In order to remove an `sfence` from a store buffer, the `sfence` must simultaneously be the left-most entry in each store buffer of that thread. When an `sfence` is removed from a store buffer, it is removed from all store buffers of that thread. By this mechanism, any store that is enqueued later than an `sfence` in any store buffer, cannot update to memory before all stores have updated to memory that were enqueued earlier than that `sfence` in any store buffer.

**Formal Semantics**

Let a program object \( P = (M^0, T, \lambda^0, \lambda^{\text{next}}, \lambda^{\text{instr}}) \) be given. In this section, we will define the state transition system \( \mathcal{T}_P^{\text{PSO}} \) induced by \( P \). The transition system is a directed, labelled graph \( \mathcal{T}_P^{\text{PSO}} = \langle S_P^{\text{PSO}}, \Delta_P^{\text{PSO}} \rangle \).

**States.**

A state \( \sigma \in S_P^{\text{PSO}} \) is a tuple \( \sigma = (M, Q, R, B) \). The elements \( M, Q \) and \( R \) are interpreted as in the SC and TSO semantics: The element \( M : \mathbb{Z} \mapsto \mathbb{Z} \) is a map from memory address to the current value held in memory at that address. For each thread \( t \in T \), we have that \( Q(t) \in \text{Labels} \) is the label of the next instruction to execute for \( t \), or \( \perp^t \) if \( t \) has terminated. For each thread \( t \in T \) and register \( r \), we have that \( R(t)(r) \in \mathbb{Z} \) is the current value of that register for \( t \).

The element \( B : T \mapsto \mathbb{Z} \mapsto (\mathbb{Z} \cup \{\text{sfence}\})^* \) represents the store buffers of each thread. For any thread \( t \) and memory address \( \alpha \), the word \( B(t)(\alpha) \) is the store buffer of \( t \) for stores to \( \alpha \). A store buffer \( B(t)(\alpha) \) is represented as a word over integers and `sfence` instructions, where each integer represents a store of that value to \( \alpha \). The `sfence` instructions are inserted into the store buffers in order to mark which stores appear before and after the fence in program order. New stores (and `sfence` instructions) are enqueued on the right side of the word, and dequeued from the left side.
The initial state is $\sigma^0_{PSO} = (\mathcal{M}^0, \lambda^0, \lambda t. \lambda r. 0, \lambda t. \lambda \alpha. \varepsilon)$.

**Transitions.**

A transition is a tuple $(\sigma_0, l, \sigma_1) \in \Delta^\text{PSO}_P$, where $\sigma_0, \sigma_1 \in \mathcal{S}^\text{PSO}_P$ are respectively the source and the target state of the transition, and $l$ is a description of how the abstract machine may transition from $\sigma_0$ to $\sigma_1$. As under TSO, $l$ may be the label of an instruction, or an update. Since each thread has multiple store buffers under PSO, in the case of an update it is necessary to indicate both the thread and the updated memory address. In case of an update, we therefore let $l$ be a pair $(t, \alpha)$ of the updating thread $t$ and the updated address $\alpha$. In addition, under PSO we have an extra transition for resolving $\textsf{sfence}$ instructions. In such case we let $l = (t, \textsf{sfence})$, where $t$ is the thread resolving the store fence. We will sometimes write $\sigma_0 \xrightarrow{l} \sigma_1$ to denote that $(\sigma_0, l, \sigma_1) \in \Delta^\text{PSO}_P$.

The set of transitions $\Delta^\text{PSO}_P$ is the smallest set such that $(\sigma_0, l, \sigma_1) \in \Delta^\text{PSO}_P$ holds with $\sigma_0 = (\mathcal{M}_0, Q_0, R_0, B_0)$ and $\sigma_1 = (\mathcal{M}_1, Q_1, R_1, B_1)$ whenever one of the following requirements holds for some thread $t$. Here we let $Q'_0$ denote the updated program counters $Q_0[t \leftarrow \lambda^{\text{next}}(l)]$ if $l \in \text{Labels}$ and $l \neq \perp^t$, and we let $Q'_0 = Q_0$ otherwise. For convenience, we use (*), to mark the rules which differ from the corresponding TSO semantics in an interesting way.

**No-op.** $l \in \text{Labels}$ and $l \neq \perp^t$ and $\lambda^{\text{instr}}(l) = \text{nop}$ and $\mathcal{M}_1 = \mathcal{M}_0$ and $Q_1 = Q'_0$ and $R_1 = R_0$ and $B_1 = B_0$.

**Local Assignment.** $l \in \text{Labels}$ and $l \neq \perp^t$ and $\lambda^{\text{instr}}(l) = (r := a)$ for some register $r$ and expression $a$, and $\mathcal{M}_1 = \mathcal{M}_0$ and $Q_1 = Q'_0$ and $R_1 = R_0$ and $B_1 = B_0$.

**Read From Memory.** $l \in \text{Labels}$ and $l \neq \perp^t$ and $\lambda^{\text{instr}}(l) = (r := [a])$ for some register $r$ and expression $a$ and furthermore there are no pending stores by $t$ to $a$, i.e., $B_0(t)(\alpha) \in \{\textsf{sfence}\}^*$, and $\mathcal{M}_1 = \mathcal{M}_0$ and $Q_1 = Q'_0$ and $R_1 = R_0[t \leftarrow R_0(t)[r \leftarrow \mathcal{M}_0(\alpha)]]$ where the loaded address is $\alpha = R_0(t)(a)$ and $B_1 = B_0$.

**Read-Own-Write-Early.** $l \in \text{Labels}$ and $l \neq \perp^t$ and $\lambda^{\text{instr}}(l) = (r := [a])$ for some register $r$ and expression $a$ and furthermore there is a most recent store of a value $v$ in $B_0(t)(\alpha)$, i.e., there is some value $v \in \mathbb{Z}$, some word $\overline{v} \in ((\mathbb{Z} \cup \{\textsf{sfence}\})^*$, and some (possibly empty) word of only store fences $\overline{\textsf{sfence}}^*$ such that $B_0(t)(\alpha) = \overline{\textsf{sfence}}^* \cdot \overline{v} \cdot \overline{a}$, and $\mathcal{M}_1 = \mathcal{M}_0$ and $Q_1 = Q'_0$ and $R_1 = R_0[t \leftarrow R_0(t)[r \leftarrow v]]$ where the loaded address is $\alpha = R_0(t)(a)$ and $B_1 = B_0$.

**Memory Store (**). $l \in \text{Labels}$ and $l \neq \perp^t$ and $\lambda^{\text{instr}}(l) = ([a_0] := a_1)$ for some expressions $a_0, a_1$, and $Q_1 = Q'_0$ and $R_1 = R_0$ and $\mathcal{M}_1 = \mathcal{M}_0$ and $B_1 = B_0[t \leftarrow B_0(t)[\alpha \leftarrow B_0(t)(a) \cdot v]]$ where $\alpha = R_0(t)(a_0)$ and $v = R_0(t)(a_1)$.
Unconditional Branch. \( l \in \text{Labels} \) and \( l \neq \bot \) and \( \lambda^{\text{instr}}(l) = (\text{goto } l') \) for some label \( l' \), and \( \mathcal{M}_1 = \mathcal{M}_0 \) and \( \mathcal{Q}_1 = \mathcal{Q}_0[t \leftrightarrow l'] \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{B}_1 = \mathcal{B}_0 \).

Conditional Branch (True). \( l \in \text{Labels} \) and \( l \neq \bot \) and for some expression \( a \) and label \( l' \) it holds that \( \lambda^{\text{instr}}(l) = (\text{if } a \text{ goto } l') \) and furthermore \( \mathcal{R}_0(t)(a) \neq 0 \), and \( \mathcal{M}_1 = \mathcal{M}_0 \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{Q}_1 = \mathcal{Q}_0[t \leftrightarrow l'] \) and \( \mathcal{B}_1 = \mathcal{B}_0 \).

Conditional Branch (False). \( l \in \text{Labels} \) and \( l \neq \bot \) and for some expression \( a \) and label \( l' \) it holds that \( \lambda^{\text{instr}}(l) = (\text{if } a \text{ goto } l') \) and furthermore \( \mathcal{R}_0(t)(a) = 0 \), and \( \mathcal{M}_1 = \mathcal{M}_0 \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{Q}_1 = \mathcal{Q}_0 \) and \( \mathcal{B}_1 = \mathcal{B}_0 \).

Full Memory Fence (*). \( l \in \text{Labels} \) and \( l \neq \bot \) and \( \lambda^{\text{instr}}(l) = \text{fence} \) and also \( \mathcal{B}_0(t)(\alpha) = \varepsilon \) for all memory addresses \( \alpha \), and \( \mathcal{M}_1 = \mathcal{M}_0 \) and \( \mathcal{Q}_1 = \mathcal{Q}_0 \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{B}_1 = \mathcal{B}_0 \).

Enqueue Store Fence (*). \( l \in \text{Labels} \) and \( l \neq \bot \) and \( \lambda^{\text{instr}}(l) = \text{sfence} \), and \( \mathcal{M}_1 = \mathcal{M}_0 \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{Q}_1 = \mathcal{Q}_0 \) and an \( \text{sfence} \) is added to all store buffers of \( t \), i.e., \( \mathcal{B}_1 = \mathcal{B}_0[t \leftrightarrow (\lambda\alpha.\mathcal{B}_0(t)(\alpha) \cdot \text{sfence})] \).

Dequeue Store Fence (*). \( l = (t, \text{sfence}) \notin \text{Labels} \) and for each memory address \( \alpha \), there is a word \( \overline{b}_0 \) such that \( \mathcal{B}_0(t)(\alpha) = \text{sfence} \cdot \overline{b}_0 \), and \( \mathcal{M}_1 = \mathcal{M}_0 \) and \( \mathcal{Q}_1 = \mathcal{Q}_0 \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{B}_1 = \mathcal{B}_0[t \leftrightarrow (\lambda\alpha.\overline{b}_0)] \).

Memory Update (*). \( l = (t, \alpha) \notin \text{Labels} \) and \( \mathcal{B}_0(t)(\alpha) = v \cdot \overline{b} \) for some value \( v \in \mathbb{Z} \) and word \( \overline{b} \), and \( \mathcal{M}_1 = \mathcal{M}_0[\alpha \leftarrow v] \) and \( \mathcal{Q}_1 = \mathcal{Q}_0 \) and \( \mathcal{R}_1 = \mathcal{R}_0 \) and \( \mathcal{B}_1 = \mathcal{B}_0[t \leftrightarrow \mathcal{B}_0(t)[\alpha \leftarrow \overline{b}]] \).

Based on this semantics, we define the concepts of a \textit{run} \( \tau \) under PSO and the reached state \( \tau(\sigma) \) analogously to how they are defined for SC and TSO.

### 3.3 POWER

IBM’s POWER architecture [26] uses a memory model which is significantly more relaxed than TSO and PSO. It allows all four basic kinds of memory access reorderings: \( W \rightarrow W \) and \( W \rightarrow R \) and \( R \rightarrow W \) and \( R \rightarrow R \). Furthermore, it allows the behaviors associated with non-atomic stores, as described in Section 3.1.3. Order between memory accesses is only maintained as a special case, between accesses to the same memory location in order to maintain coherence, between memory accesses that are dependent on each other, and between memory accesses separated by fences.

The POWER memory model is described informally in the architecture manual [26]. There have since been several proposals for formalization, e.g. in [12, 46, 14]. These are in many cases supported by manual interpretation of the architecture manual, and extensive testing of the hardware. In this section, we will recall the axiomatic model introduced by Alglave et al. in [14].
3.3.1 Intuition of POWER

Here we give a very informal overview of the POWER memory model. To gain a working understanding of the model it is necessary to depend on the complete formalization.

The POWER memory model allows all the memory relaxations described in Section 3.1.3. It does however maintain the coherence property, described in Section 3.1.4. Sarkar et al. suggest in [46] that the POWER memory model’s behavior can be explained by each thread keeping track of the order of all memory stores which they have observed, and propagating their own stores to each other thread separately at nondeterministic times.

**Example 3.9 (MP+sync+addr).** Figure 3.23 shows a Shasha-Snir trace in another variation of the MP idiom. The trace contains a cycle corresponding to a non-SC behavior, since the loads M0 and M2 observe the stores L0 and L2 in the wrong order. However, this behavior is not allowed (i.e., not possible) under the POWER memory model. This is because the order between L0 and L2, and between M0 and M2 are enforced by a sync instruction and an address dependency respectively. Notice that there is an address dependency from M0 to M2, since there is a data flow within the same thread from the value read by M0 to the computation of the address accessed by M2. This data flow induces an address dependency despite the fact that the actual value in the register $r0$ does not affect the accessed address ($r1$ will hold the value 0 regardless).

Dependencies.

There are three main dependencies which enforce order between memory accesses under POWER: address, data and control dependencies.

A memory accessing instruction $i$ is *address dependent* on a program order-earlier instruction $i'$ when $i'$ produces a value in a register which is fed into the computation of the address accessed by $i$. Notice however, that the data
flow making up an address dependency may only include data flow within
the same thread, and which is visible statically in the program code. I.e., the
data flow may contain chains of instructions which put values into registers,
which are then used by the subsequent instructions. But a data flow from a
store instruction to the load instruction which reads its written value does not
contribute to forming an address dependency, since it is not statically visible.

A data dependency is like an address dependency, but the value used by the
dependent instruction is used for something other than computing a memory
address. For example, if a load instruction \(i\) puts the read value into a reg-
ister \(r\), and a subsequent store instruction \(i'\) stores the value of \(r\) into some
shared variable, then there is a data dependency from \(i\) to \(i'\). In Figure 3.23,
there are address dependencies to \(M2\) from both \(M0\) and \(M1\), and there is a data
dependency from \(M0\) to \(M1\).

An instruction \(i\) is control dependent on a program order-earlier instruction
\(i'\) if there is a conditional branch instruction \(i''\) which program order-earlier
than \(i\) and which is data dependent on \(i'\). For example, in Figure 3.7 the con-
ditional branch instruction \(M1\) is data dependent on the load \(M0\). Therefore the
program order-later load \(M2\) is control dependent on \(M0\). As we will see later,
under POWER control dependencies enforce memory access order only from a
load to a control dependent store. Not from a load to a control dependent load.
This is why the non-SC trace shown in Figure 3.8 is allowed under POWER.

Fences.
The POWER architecture supports several kinds of fences. The three kinds
that we will consider here are sync, lwsync and isync.

The sync fence is the strongest fence under POWER. It will enforce all
four kinds of memory access orderings: \(W \rightarrow W\) and \(W \rightarrow R\) and \(R \rightarrow W\)
and \(R \rightarrow R\). For example, in Figure 3.23, the sync instruction serves to
enforce order between the stores \(L0\) and \(L2\). Furthermore, sync fences have a
cumulative effect. This means that when a thread \(t\) executes a sync, not
only is order enforced between accesses \(i_0\) preceding the sync and accesses \(i_1\)
succeeding the sync in program order. Additionally, other memory accesses
performed by other threads \(t'\), which have been observed by \(t\) will also become
ordered with \(i_1\), and this order will be observable by third threads \(t''\).

Example 3.10 (Cumulativity). Let us consider the Shasha-Snir trace in Fig-
ure 3.24. It shows a non-SC behavior which is allowed under POWER. We
see that the thread \(Q\) reads the value written to \(x\) by the store \(L0\). Then \(Q\)
passes the value along to \(M2\), after which it is read by the load \(N0\) in the
thread \(R\). The value is then used to compute the address (of the variable
\(x\)) accessed by the load \(N1\). Despite this apparent causality, the load \(N1\)
is able to read the initial value of \(x\), and not observe the store \(L0\). Notice
that this happens even though all pairs of instructions in the same thread
have dependencies between each other. One could intuitively explain this
behavior in the operational semantics of Sarkar et al. [46], by saying that the thread P propagates the store L0 to Q early, but does not propagate the store to R until much later.

Here, the cumulativity of a sync fence may serve to restore order. Consider the Shasha-Snir trace in Figure 3.25. It is the same as Figure 3.24, except that the data dependency between M0 and M2 has been replaced by a sync. In this case, the behavior is forbidden under POWER. I.e., it cannot happen. Intuitively, this is because L0 was observed by M0, before the sync, and therefore becomes, cumulatively, ordered before M2. So when R observes M2, it must also observe L0. Hence, in this case N1 would have to read the value written by L0, rather than the initial value.

A lwsync fence (“light-weight sync”) is similar to a sync, but it does not enforce the $W \rightarrow R$ memory access ordering. Furthermore, its cumulativity is slightly weaker than that of a full sync.

We noted earlier that control dependencies do not enforce order between loads (only from load to store). The isync fence serves to strengthen a control dependency such that it does enforce order between loads. Specifically, if a load $i$ is program order-after an isync, which is program order-after a conditional branch $i'$, which is data dependent on a program order-earlier load
then order is enforced between \( i'' \) and \( i \). We say then that there is a control + isync dependency between \( i'' \) and \( i \).

**Example 3.11 (MP+lwsync+ctrl+isync).** In Figure 3.26, a lwsync enforces order between the stores \( L_0 \) and \( L_2 \). There is a control dependency between the loads \( M_0 \) and \( M_3 \). The control dependency in itself is not sufficient to enforce order between \( M_0 \) and \( M_3 \), but together with the isync fence it suffices.

### 3.3.2 Shasha-Snir Traces: Formalization and Derived Relations

As mentioned above axiomatic memory models are predicates over Shasha-Snir traces, deciding for each behavior whether or not it is allowed. Here we will first formalize the notion of Shasha-Snir trace, following [14]\(^2\). Then we will define various derived relations and functions working on Shasha-Snir traces. These will be used in the following section for defining the POWER memory model.

**Shasha-Snir Traces: Formalization**

Let a program \( P = (\mathcal{M}^0, T, \lambda^0, \lambda^{\text{next}}, \lambda^{\text{instr}}) \) be given. A Shasha-Snir trace induced by a run of the program \( P \) is a quadruple \((E, \text{po}, \text{co}, \text{rf})\). Here, \( E \) is a set of events, and \( \text{po}, \text{co}, \text{rf} \in E \times E \) are relations over events. An event represents the execution of a program instruction by some thread. Formally, an event \( e \in E \) is a tuple \( e = (l, n) \), where \( l \) is the label of the executed instruction, and \( n \) is a natural number which distinguishes between multiple events corresponding to different executions of the same instruction. In particular, we

\(^2\)In [14], the authors use the name *executions* for Shasha-Snir traces.
choose \( n \) to be the position of the event in the program order of its thread. I.e., the program order-earliest event of each thread has the index 0, the next event has index 1, etc.

In addition to events corresponding to executed instructions, \( E \) also contains special initializer events, which are implicit stores which provide each memory location with its initial value. For each memory address \( \alpha \), the initializer event \((l_\alpha, 0)\) is contained in \( E \). Here \( l_\alpha \) is assumed to be a unique label which does not appear in \( \mathcal{P} \).

For an event \( e = (l, n) \) we define \( \text{label}(e) = l \). Furthermore, we let \( \text{tid}(e) \) denote the thread executing the event. I.e., \( \text{tid}(e) \) is the unique thread \( t \) such that \( l \) is reachable from \( \lambda^0(t) \) in \( \mathcal{P} \). If \( e \) is an initializer event \((l_\alpha, 0)\) for some memory address \( \alpha \), then we let \( \text{tid}(e) = t^\text{init}_\alpha \), where \( t^\text{init}_\alpha \) is some unique thread identifier which does not appear in \( \mathcal{T} \). We also let \( \text{instr}(e) \) denote the executed instruction \( \lambda^\text{instr}(l) \). If \( e \) is an initializer event \((l_\alpha, 0)\) for some memory address \( \alpha \), then we let \( \text{instr}(e) = ([\alpha] : = \mathcal{M}^0(\alpha)) \) be a store which gives that memory location its initial value.

The relation \( \text{po} \) is the program order relation. Formally \( \text{po} \) contains \((e, e')\) for two events \( e = (l, n) \) and \( e' = (l', n') \) precisely when \( \text{tid}(e) = \text{tid}(e') \) and \( n < n' \). The relation \( \text{co} \) is the coherence order. It relates stores to the same memory location, and is a total order when restricted to stores to any one memory location. The relation \( \text{rf} \) is the read from relation. It contains pairs \((e, e')\) such that \( e \) is a store to some memory location \( x \), and \( e' \) is a load from \( x \) which reads the value written by \( e \).

**Derived Relations and Functions**

Here we will define a number of derived relations over the events in a Shasha-Snir trace. These will allow us to define what it means for a Shasha-Snir trace to be well-formed, and eventually set the stage for us to define the axiomatic model of POWER. Throughout this section, we let a Shasha-Snir trace \(\pi = (E, \text{po}, \text{co}, \text{rf})\) be fixed.

**Kinds of Memory Accesses.**

It will sometimes be necessary to restrict various relations depending on which kinds of memory accesses they relate. For example, we might restrict a relation to only the cases where e.g. the first element is a store and the second a load. To facilitate this, we define the four relations \( \text{RR}_\pi \), \( \text{RW}_\pi \), \( \text{WR}_\pi \) and \( \text{WW}_\pi \) as follows:

\[
\begin{align*}
\text{R}_\pi &= \{ e \in E \mid \text{instr}(e) \text{ is a load} \} \\
\text{W}_\pi &= \{ e \in E \mid \text{instr}(e) \text{ is a store} \} \\
\text{RR}_\pi &= \text{R}_\pi \times \text{R}_\pi \\
\text{RW}_\pi &= \text{R}_\pi \times \text{W}_\pi \\
\text{WR}_\pi &= \text{W}_\pi \times \text{R}_\pi \\
\text{WW}_\pi &= \text{W}_\pi \times \text{W}_\pi
\end{align*}
\]
Communication Relations.
The relations $\text{co}$ and $\text{rf}$ tell us how the memory accesses in the Shasha-Snir trace interact. In order to reason about this interaction it is useful to have a couple of refined, derived relations as well.

The first is the $\text{fr}_\pi$ (“from read”) relation, which relates each load to all stores which come later than the source store of that load in coherence order. I.e. it relates a load to all stores which overwrite the value that was read by the load. Formally, we define $\text{fr}_\pi = \{(e, e') | \exists e_w. (e_w, e) \in \text{rf} \land (e_w, e') \in \text{co}\}$.

Sometimes it is useful to distinguish between communication within the same thread, and communication between different threads. Therefore we split the relation $\text{rf}$ into two relations $\text{rfe}_\pi$ and $\text{rfi}_\pi$ for external and internal read from respectively. Formally we have,

\begin{align*}
\text{rfe}_\pi &= \{(e, e') | \text{rf}(\text{tid}(e) \neq \text{tid}(e'))\} \\
\text{rfi}_\pi &= \{(e, e') | \text{rf}(\text{tid}(e) = \text{tid}(e'))\}
\end{align*}

In the same way, we split $\text{co}$ into $\text{coe}_\pi$ and $\text{coi}_\pi$, and $\text{fr}_\pi$ into $\text{fre}_\pi$ and $\text{fri}_\pi$.

Furthermore, for convenience, we define the union of all communication relations: $\text{com}_\pi = \text{co} \cup \text{rf} \cup \text{fr}_\pi$.

Dependencies.
First we define the data dependency relation $\text{data}_\pi \subseteq \text{po}$. The data dependency relation is the smallest transitive relation such that $(e_0, e_1) \in \text{data}_\pi$ whenever there are registers $r, r_1$, and expressions $a_0, a_1, a'_1$, and a label $l_1$ such that $\text{instr}(e_1)$ is either $(r_1 := a_1)$ or $([a'_1] := a_1)$ or $(\text{if } a_1 \text{ goto } l_1)$, and the expression $a_1$ contains the register $r$, and $\text{instr}(e_0)$ is either $(r := a_0)$ or $(r := [a_0])$.

We can then define the address dependency relation $\text{addr}_\pi \subseteq \text{po}$. The address dependency relation $\text{addr}_\pi$ is the smallest relation such that

- $(e_0, e_1) \in \text{addr}_\pi$ when there are registers $r, r_1$ and expressions $a_0, a_1, a'_1$ such that $\text{instr}(e_1)$ is either $(r_1 := [a_1])$ or $([a'_1] := a_1)$ and the expression $a_1$ contains the register $r$ and $\text{instr}(e_0)$ is either $(r := a_0)$ or $(r := [a_0])$, and
- $(e_0, e_1) \in \text{addr}_\pi$ when there is another event $e$ such that $(e_0, e) \in \text{data}_\pi$ and $(e, e_1) \in \text{addr}_\pi$.

We also define the control dependency relation $\text{ctrl}_\pi \subseteq \text{po}$. The control dependency relation $\text{ctrl}_\pi$ is the smallest relation such that

- $(e_0, e_1) \in \text{ctrl}_\pi$ when $\text{instr}(e_0)$ is a conditional or unconditional branch instruction, and
- $(e_0, e_1) \in \text{ctrl}_\pi$ when there is another event $e$ such that $(e_0, e) \in \text{data}_\pi$ and $(e, e_1) \in \text{ctrl}_\pi$.

Fences.
In order to easily reason about which events are separated in the control flow by a fence, we introduce the relations $\text{sync}_\pi$, $\text{lwsync}_\pi$ and $\text{isync}_\pi$, capturing
respectively the events separated by a \texttt{sync}, \texttt{lwsync} or \texttt{isync} fence. More precisely, \((e, e') \in \text{sync}_\pi\) iff \((e, e') \in \text{po}\) and either (i) \(\text{instr}(e) = \text{sync}\), or (ii) \(\text{instr}(e') = \text{sync}\), or (iii) there is another event \(e''\), such that \((e, e'') \in \text{po}\) and \((e'', e') \in \text{po}\) and \(\text{instr}(e'') = \text{sync}\). The relations \text{lwsync}_\pi\) and \text{isync}_\pi\) are defined correspondingly.

**Computation of Values and Addresses.**
The flow of data through registers in a thread is visible in the instructions of the events of a Shasha-Snir trace. In order to find the values held by registers at the time when an event is executed, we can therefore recursively follow this flow back to where the registers were previously assigned. A requirement for this to be possible is that the data flow is not cyclic. I.e., that the relation \text{data}_\pi \cup \text{addr}_\pi \cup \text{rf} is acyclic. A second requirement is that for each load event \(e_r \in E\) there is precisely one event \(e_w \in E\) such that \((e_w, e_r) \in \text{rf}\). Furthermore, the event \(e_w\) should be a store event. Under these assumptions, we define the function \(\text{value}_\pi(e, a)\) which computes the value of the expression \(a\) when evaluated at the event \(e\) in the Shasha-Snir trace \(\pi\):

- If \(a \in \mathbb{Z}\), then \(\text{value}_\pi(e, a) = a\).
- If \(a = f(a_0, \ldots, a_n)\) for some operator \(f\) and subexpressions \(a_0, \ldots, a_n\), then \(\text{value}_\pi(e, a) = f(\text{value}_\pi(e, a_0), \ldots, \text{value}_\pi(e, a_n))\).
- If \(a = r\) for some register \(r\), then let \(e'\) be the program order-latest event such that \((e', e) \in \text{po}\) and there is some expression \(a'\) such that either \(\text{instr}(e') = (r := a')\) or \(\text{instr}(e') = (r := [a']\).
  - If there is no such event \(e'\), then \(\text{value}_\pi(e, a) = 0\).
  - If \(\text{instr}(e') = (r := a')\), then \(\text{value}_\pi(e, a) = \text{value}_\pi(e', a')\).
  - If \(\text{instr}(e') = (r := [a']\), then let \(e_w\) be the store event such that \((e_w, e') \in \text{rf}\), and let \(a_a, a_d\) be the expressions such that we have \(\text{instr}(e_w) = ([a_a] := a_d)\). Then \(\text{value}_\pi(e, a) = \text{value}_\pi(e_w, a_d)\).

With this definition, we can easily define the function \(\text{address}_\pi(e)\), which gives the address accessed by a memory access \(e\): If \(\text{instr}(e) = (r := [a_a])\) or \(\text{instr}(e) = ([a_a] := a_d)\) for some expressions \(a_a, a_d\) and some register \(r\), then we define \(\text{address}_\pi(e) = \text{value}_\pi(e, a_a)\). If \(e\) is not a memory accessing instruction, then we let \(\text{address}_\pi(e) = \perp\).

**Program Order per Location.**
We define program order per location \(\text{po-loc}_\pi \subseteq \text{po}\) to be the program order relation restricted to memory access events targeting the same memory address:
\[
\text{po-loc}_\pi = \{ (e, e') \in \text{po} | \text{address}_\pi(e) = \text{address}_\pi(e') \neq \perp \}.
\]

**Well-Formedness of Shasha-Snir Traces.**
We are now ready to define a number of basic requirements for a Shasha-Snir trace to intuitively make sense. We define here the notion of \textit{well-formedness}
for Shasha-Snir traces. It will be used as a minimum requirement for the memory model to allow a certain behavior.

**Definition 3.1 (Well-Formedness of Shasha-Snir Traces).** We say that a trace \( \pi = (E, po, co, rf) \) is well-formed if all of the following requirements hold:

1. \( po = \{(e, e')| tid(e) = tid(e') \wedge n < n' \text{ where } e = (l, n) \wedge e' = (l', n')\} \),
2. for each pair \( (e, e') \in co \) it holds that both \( e \) and \( e' \) are memory stores,
3. for each pair \( (e, e') \in rf \) it holds that \( e \) is a store and \( e' \) is a load,
4. for each load \( e' \in E \) there is exactly one event \( e \in E \) such that \( (e, e') \in rf \),
5. the relation \( data_{\pi} \cup addr_{\pi} \cup rf \) is acyclic,
6. for each pair \( (e, e') \in co \cup rf \) it holds that \( address_{\pi}(e) = address_{\pi}(e') \),
7. \( co \) is a partial order such that for all distinct store events \( e, e' \in E \) it holds that \( address_{\pi}(e) = address_{\pi}(e') \) if and only if either \( (e, e') \in co \) or \( (e', e) \in co \),
8. for each event \( e \in E \) we have either \( tid(e) \in T \) (i.e., \( e \) is an ordinary event), or \( tid(e) = t_{\text{init}} \) for some address \( \alpha \) (i.e., \( e \) is an initializer event),
9. for each address \( \alpha \in \mathbb{Z} \) there is precisely one event \( e_{\alpha} \in E \) such that \( tid(e_{\alpha}) = t_{\text{init}}^{\alpha} \), and for that event we have \( e_{\alpha} = (l, 0) \) for some label \( l \) and \( instr(e_{\alpha}) = ([\alpha] := M^0(\alpha)) \), and
10. for each thread \( t \in T \) there are some events \( e_0 = (l_0, 0), e_1 = (l_1, 1), \ldots, e_n = (l_n, n) \) such that \( \{e \in E| tid(e) = t\} = \{e_0, e_1, \ldots, e_n\} \) and for each index \( 0 \leq i < n \) we have either
   - \( \lambda_{\text{next}}(l_i) = l_{i+1} \) and \( instr(e_i) \) is not a branch instruction, or
   - \( instr(e_i) = (\text{goto } l_{i+1}) \), or
   - \( instr(e_i) = (\text{if } a \text{ goto } l_{i+1}) \) for some expression \( a \) and it holds that \( value_{\pi}(e_i, a) \neq 0 \), or
   - \( \lambda_{\text{next}}(l_i) = l_{i+1} \) and \( instr(e_i) = (\text{if } a \text{ goto } l) \) for some expression \( a \) and label \( l \) and it holds that \( value_{\pi}(e_i, a) = 0 \).

### 3.3.3 The POWER Memory Model

We are now ready to define the axiomatic POWER memory model \( M^{POWER} \) according to [14]. The predicate \( M^{POWER} \) holds for a Shasha-Snir trace \( \pi = (E, po, co, rf) \) iff all of the following requirements hold:

**WELL-FORMEDNESS.** The trace \( \pi \) is well-formed.

**SC PER LOCATION.** The relation \( po-loc_{\pi} \cup com_{\pi} \) is acyclic.

**NO THIN AIR.** The relation \( hb_{\pi} \) is acyclic.

**OBSERVATION.** The relation \( fre_{\pi}; prop_{\pi}; hb_{\pi}^* \) is irreflexive.

**PROPAGATION.** The relation \( prop_{\pi} \cup co \) is acyclic.

Here we have the following definitions:
\[
\text{hb}_{\pi} = \text{ppo}_{\pi} \cup \text{fences}_{\pi} \cup \text{rfe}_{\pi}
\]
\[
\text{fences}_{\pi} = \text{sync}_{\pi} \cup (\text{lwsync}_{\pi} \setminus \text{WR}_{\pi})
\]

The relations \text{ppo}_{\pi} and \text{prop}_{\pi} are discussed next.

**Preserved Program Order (ppo\textsubscript{\pi}).**

The preserved program order relation \text{ppo}_{\pi} formalizes the hitherto informal notion of which dependencies enforce order between memory accesses.

The main intuition behind the definition of \text{ppo}_{\pi} is the following. Each event is considered to be executed in two steps: initialization and completion. For a load, the initialization can be thought of as the time when the value is read from memory, and the completion can be thought of as the time when the instruction is committed. For a store, initialization is when the instruction is committed, and completion is when the written value is first made available to another thread. Different kinds of relations between events will enforce restrictions on different steps of the events.

For example, when a store \(e_w\) and a program order-later load \(e_r\) by the same thread, are related by \text{rfi}_{\pi} (i.e. the load reads the value from the store), this forces their initialization steps to occur according to program order. This is because the value of the store becomes available to the same thread at the initialization step of \(e_w\), and this value is needed when the load reads the value at the initialization step of \(e_r\). One may think of this as \(e_w\) entering the thread-local store buffer at its initialization. The pending store may be read by the load \(e_r\) of the same thread before the store completes, i.e. before the completion step of \(e_w\). However, if \(e_w\) and \(e_r\) are related by the relation \text{coe}_{\pi}; \text{rfe}_{\pi}, i.e., \(e_r\) reads from another store \(e'_w\) by another thread, which is coherence-after \(e_w\), then the completion step of \(e_w\) is forced to occur before the initialization step of \(e_r\). This is intuitively because the load \(e_r\) cannot read the value from another thread as long as \(e_w\) remains in the thread-local store buffer. So the initialization step of \(e_r\) must then happen after the completion step of \(e_w\).

To formalize this, we introduce the relations \(\text{ii}_{\pi}, \text{ic}_{\pi}, \text{ci}_{\pi}\) and \(\text{cc}_{\pi}\) which are the relations enforcing order on respectively initialization step to initialization step, initialization step to completion step, completion step to initialization step and completion step to completion step between different events in the same thread. All four relations are limited to events in the same thread: \(\text{ii}_{\pi}, \text{ic}_{\pi}, \text{ci}_{\pi}, \text{cc}_{\pi} \subseteq \text{po}\).

First we define a few basic relations:

\[
\text{rdw}_{\pi} = \text{po-loc}_{\pi} \cap (\text{fre}_{\pi}; \text{rfe}_{\pi})
\]
\[
\text{detour}_{\pi} = \text{po-loc}_{\pi} \cap (\text{coe}_{\pi}; \text{rfe}_{\pi})
\]
\[
\text{ii}^0_{\pi} = \text{addr}_{\pi} \cup \text{data}_{\pi} \cup \text{rdw}_{\pi} \cup \text{rfi}_{\pi}
\]
\[
\text{ci}^0_{\pi} = (\text{ctrl}_{\pi}; \text{isync}_{\pi}) \cup \text{detour}_{\pi}
\]
\[
\text{cc}^0_{\pi} = \text{addr}_{\pi} \cup \text{data}_{\pi} \cup \text{po-loc}_{\pi} \cup \text{ctrl}_{\pi} \cup (\text{addr}_{\pi}; \text{po})
\]
We can then define $ii_\pi$, $ic_\pi$, $ci_\pi$ and $cc_\pi$, as the smallest relations satisfying the equations below. The intuition here is that $ii_\pi$, $ci_\pi$ and $cc_\pi$ contain respectively $i_{\pi}^0_\pi$, $c_{\pi}^0_\pi$ and $cc_{\pi}^0_\pi$. Furthermore $ii_\pi$ contains any chain of enforced order between the various initialization and completion steps, provided that the chain forces order between the initialization steps of the first and last event in the chain. E.g. $ic_\pi; cl_\pi$ is contained in $ii_\pi$, capturing chains from initialization step to initialization step via some completion step. The relation $ci_\pi$ is contained in $ii_\pi$ since for any $(e, e') \in cl_\pi$, the initialization step of the first event $e$ is ordered before the completion step of the same event $e$ by the instruction semantics, and the completion step of $e$ is ordered before the initialization step of $e'$ by $ci_\pi$. The same intuition explains the remaining definitions as well.

$$
ii_\pi = i_{\pi}^0_\pi \cup cl_\pi \cup (ic_\pi; cl_\pi) \cup (ii_\pi; ii_\pi) \\
ic_\pi = i_{\pi}^0_\pi \cup cc_\pi \cup (ic_\pi; cc_\pi) \cup (ii_\pi; ic_\pi) \\
ci_\pi = c_{\pi}^0_\pi \cup (ci_\pi; ii_\pi) \cup (cc_\pi; ci_\pi) \\
cc_\pi = cc_{\pi}^0_\pi \cup ci_\pi \cup (ci_\pi; ic_\pi) \cup (cc_\pi; cc_\pi)
$$

And finally, we can define the preserved program order relation as the relation from loads to loads or stores where order is enforced between the initialization step of the loads, and the completion step of the stores:

$$ppo_\pi = (ii_\pi \cap RR_\pi) \cup (ic_\pi \cap RW_\pi)$$

**Example 3.12** (Preserved Program Order Chains). Consider the Shasha-Snir trace shown in Figure 3.27. There is a control dependency from the load $L0$ to the store $L2$. Since a control dependency suffices to order the completion step of the store after the initialization step of the load (formally $ctrl_\pi \in cc_{\pi}^0_\pi \subseteq cc_\pi \subseteq ic_\pi$), we have that $(L0, L2) \in ppo_\pi$. I.e., program order is preserved between $L0$ and $L2$. 

---

**Figure 3.27.** A Shasha-Snir trace illustrating a few derived relations which induce preserved program order ($ppo_\pi$) relationships between events.
The same control dependency exists from L₀ to the load L₃, so we have (L₀, L₃) ∈ icᵣ. However, as we can see in the definition of ppoᵣ, enforcing order from the initialization step of a load to the completion step of a later load is not sufficient to preserve program order. So the control dependency does not suffice here to enforce order between L₀ and L₃.

However, since the load L₃ reads the value written by the store M₀, its initialization step must occur later than the completion step of the earlier store L₂ to the same memory location (according to the reasoning above). Formally (L₂, L₃) ∈ detourᵣ ⊆ crᵣ, ⊆ ciᵣ. Therefore, the initialization step of L₃ must also be ordered after the initialization step of L₀. Formally we have (L₀, L₂) ∈ icᵣ and (L₂, L₃) ∈ ciᵣ, and so we have the chain (L₀, L₃) ∈ (icᵣ; ciᵣ) ⊆ iiᵣ. Since the initialization steps of L₀ and L₃ are kept in order, we also have that program order is preserved between L₀ and L₃, i.e., (L₀, L₃) ∈ ppoᵣ.

Notice however that this would not be the case if L₃ had read the value from the store L₂ instead of M₀.

**Propagation (propᵣ).**

The propagation relation propᵣ intuitively keeps track of the way fences affect the order in which stores may be observed by different threads – the order in which the stores are propagated to the different threads. This includes capturing the effects of cumulativity, as informally discussed earlier. Notice that propagation is affected by sync and lwsync, but not by isync. The definition of propᵣ is relatively succinct (compared to ppoᵣ):

\[
propᵣ = \left( \begin{array}{c} \text{prop-base} \cap WWᵣ \\ \text{com} \cap \text{prop-base} \cap sync \cap \text{hb} \\
\end{array} \right)
\]

\[
\text{prop-base} = \text{rfe} \cup \text{fences} \cup \text{hb}
\]

\[
\text{fences} = \text{sync} \cup \text{lwsync} \cup \text{WR}
\]
**Example 3.13** (Propagation in WRC). Consider again the cumulativity example in Figure 3.25, which is repeated for convenience in Figure 3.28. Here we will see that there is a propagation relationship from the store \(L_0\) to the store \(M_2\). This is because the \(rfe\) edge from \(L_0\) to \(M_0\), together with the \(sync\) edge from \(M_0\) to \(M_2\) constitute a \(prop\)-base edge. And since both \(L_0\) and \(M_2\) are stores, we have \((L_0, M_2) \in (prop\)-base \(\cap WW) \subseteq prop\).

In the same figure, the entire cycle from \(N_1\) and back to \(N_1\) is actually a reflexive edge in \(prop\_\pi\). To see this, note that the \(fr\); \(rf\)-edge from \(N_1\) to \(M_0\) is in \(com\_\pi\). There is a \(sync\_\pi\) edge from \(M_0\) to \(M_2\). And the \(rf\); \(addr\_\pi\) relation from \(M_2\) back to \(N_1\) constitute an \(hb\_\pi\)-edge.

The propagation edge from \(L_0\) to \(M_2\) would remain if the \(sync\) were replaced by a \(lwsync\), but the edge from \(N_1\) back to \(N_1\) would disappear.

### 3.3.4 POWER Examples Explained

The previous section presented the axiomatic model of POWER. Here we will have a look at a number of example traces, and explain why they are allowed or forbidden by the axiomatic rules presented above. This aims to solidify the reader’s understanding of the rules.

**Example 3.14** (SC PER LOCATION). The coherence criterion described earlier, in Section 3.1.4, is enforced under POWER directly by the SC PER LOCATION requirement. Consider both examples of violations of the coherence criterion given in Figure 3.13. Notice that each of them has a cycle in the relation \(com\_\pi \cup po\_loc\_\pi\).

**Example 3.15** (NO THIN AIR). Figure 3.29 shows a behavior which is forbidden under POWER. We see that each of the loads \(L_0\) and \(M_0\) reads the value written by the other thread, and that the very reading of those values is what causes control flow to reach those stores. Formally, this
behavior is forbidden under POWER, by the NO THIN AIR requirement. This is because the control dependencies between \textbf{L0} and \textbf{L2} as well as \textbf{M0} and \textbf{M2}, together with the read from edges form a cycle in the \textit{hb} relation.

If any of the control dependencies were removed, and replaced with only program order, then the behavior would be allowed under POWER.

\textbf{Example 3.16 (OBSERVATION).} We consider again the example in Figure 3.26, repeated for convenience in Figure 3.30. This behavior is forbidden under POWER because of the OBSERVATION requirement. Notice that there is a reflexive edge in \textit{fre}_\pi; \textit{prop}_\pi; \textit{hb}^* starting and ending in \textbf{M3}. The \textit{fre}_\pi relation takes us from \textbf{M3} to \textbf{L0}. From there, the \textit{prop}_\pi relation leads to \textbf{L2}. And from there \textit{hb}_\pi takes us back to \textbf{M3}. Notice that the \textit{prop}_\pi relation only relates stores, in the absence of a full \textit{sync} fence. That is the reason why the \textit{prop}_\pi relation can take us only to \textbf{L2}, and not all the way back to \textbf{M3}.

\textbf{Example 3.17 (PROPAGATION).} Figure 3.31 shows again the IRIW idiom, which showcases the non-atomic writes of POWER. Here, two \textit{sync} fences have been added to the threads \textbf{R} and \textbf{S}. These suffice to make the behavior forbidden under POWER. In particular, the behavior is forbidden by the PROPAGATION requirement. To see this, notice that the \textit{fr}_\pi; \textit{rf}_\pi edge from \textbf{M2} to \textbf{N0} is an edge in \textit{com}_\pi^*. Together with the \textit{sync}_\pi edge from \textbf{N0} to \textbf{N2}, it forms a \textit{prop}_\pi edge from \textbf{M2} to \textbf{N2}. There is another, symmetric, \textit{prop}_\pi edge from \textbf{N2} to \textbf{M2}. These edges form a forbidden cycle in \textit{co} \cup \textit{prop}_\pi.
Notice that lwsync fences are not sufficient here, since they do not allow us to include the fr\textsubscript{r} edge in the cycle. Introducing some instance of preserved program order between the loads in threads R and S is also insufficient, since preserved program order without fences does not enforce any cumulative order on the stores appearing in the threads P and Q. Here, only the cumulative strength of the full sync fences is enough to enforce SC behavior.
Stateless model checking (SMC) [22] is a technique for systematic testing. With SMC, concurrent programs are tested under different schedules in a systematic way such that one can guarantee to cover all program behaviors up to a given bound. In this chapter, we will first discuss the basic operation of SMC, its advantages and disadvantages, as well as solutions to some of the disadvantages. Then, in Sections 4.2 and 4.3, we will examine and gain intuition of how the papers III and IV extend stateless model checking to support programs running under relaxed memory models.

**SMC for SC**

In Section 2.2, we saw how the semantics of a program under sequential consistency can be formalized as a transition system. With traditional explicit state model checking (such as in e.g. the SPIN model checker [25]), an analysis tool explicitly constructs and analyzes this transition system, in order to identify errors in the program. This requires the analysis tool to keep in memory a representation of all states in the transition system. For larger programs, the number of states typically grows prohibitively large, and the representation may not fit in memory. The stateless model checking technique offers a way to explore the transition system of a program without storing a representation of all states in memory. Instead of exploring the state space by building the transition system, SMC explores the state space by exploring runs of the program up to some predetermined bound. The runs through a transition system can be explored systematically such that at each point in time, it is only necessary to keep one run at a time in memory. This brings down the storage requirements from the size of the entire transition system to some constant, proportional to the predetermined bound on the lengths of the explored runs.

In its basic form, SMC is then a trade-off where a small memory footprint is bought at the price of exploring a possibly large number of program runs, where the same state may appear multiple times. For a naive implementation of SMC, the number of runs may be crippling, since it is exponential in the
Figure 4.1. A small program with 12 different runs under SC.

length of the runs. This problem has been addressed by partial order reduction (POR) [54, 42, 23, 19] techniques, adapted to SMC as dynamic partial order reduction (DPOR) [21, 47, 32, 45, 53, 1].

Partial order reduction techniques are based on the idea that a run can be considered as a partially ordered set of events, where each event corresponds to an executed instruction and is ordered with other, conflicting events. Events are considered to be conflicting when the order in which they appear in a run is somehow observable. This typically means events which participate in a data race. Such a partial order over events is known as a happens-before order (not to be confused with the hb relation of POWER), since it intuitively shows which events can be observed to happen before others. Since many runs will induce the same happens-before order, the orders define equivalence classes over the runs. These classes are known as Mazurkiewicz traces [39]. Since each run in a Mazurkiewicz trace resolves conflicts by ordering the conflicting events in the same way, all of the runs have the same observable behavior. Hence, it is sufficient for an analysis technique to explore only one run per Mazurkiewicz trace. Dynamic partial order reduction for stateless model checking strives to minimize the number of explored runs, while guaranteeing to explore at least one run per Mazurkiewicz trace.

Example 4.1 (Mazurkiewicz Traces). In order to make the intuition of Mazurkiewicz traces concrete, let us consider the small program shown in Figure 4.1. Under sequential consistency, it has 12 distinct runs, corresponding to the 12 different ways to interleave the four instructions. However, most of these runs do not have unique behaviors. Figure 4.2 shows the four Mazurkiewicz traces of the program. They capture all the observable behaviors of the program without keeping track of unobservable details such as the internal order of events which are not in conflict. Hence, to explore all observable behaviors of the program, it would suffice to explore four different runs, rather than all 12.

Here we use the convention that black arrows indicate pairs of events which are ordered in the partial order, and which cannot be reordered. We use red arrows for conflicts, where the ordering of the events might be reversible. In the case of SC, black arrows then correspond to program order, while red arrows correspond to data races.
Figure 4.2. The four Mazurkiewicz traces for the program in Figure 4.1 under SC.

Context for SMC

Stateless model checking is usually considered in the context of programs which are parallel and deterministic in the sense that they have fixed input and don’t contain any data non-determinism. SMC requires that the target program has a finite number of runs, and that each run is terminating. For some programs this holds without any modification. For other programs it is necessary to apply a bound of some sort. Such a bound is typically a fixed maximum number of allowed iterations per loop, or a maximum number of executed instructions per thread and run.

Since SMC considers only finite runs, it is best suited to finding violations of safety properties, as opposed to liveness properties. The desired properties may be expressed by specifications, or as the absence of certain runtime errors, such as assertion violations, null pointer dereferencing, buffer overflow etc.

Due to the particular strengths and limitations of SMC, the technique is especially useful in cases where a parallel algorithm is applied to a given restricted scenario. For example, given a parallel unit test, SMC is well suited to answer the question whether the unit test can fail for any thread scheduling.

Papers III and IV extend the use of DPOR from sequential consistency to work also for the relaxed memory models TSO, PSO (Paper III) and POWER (Paper IV). In this chapter, we will first examine one recent technique for SMC with DPOR under SC in Section 4.1, and then see how Papers III and IV extend its use to TSO and PSO in Section 4.2 and to POWER in Section 4.3.
4.1 Source-DPOR

Before discussing my extensions of SMC/DPOR to relaxed memory, we need to have an understanding of the earlier techniques for SMC/DPOR under SC upon which the extensions build. In this section we will discuss the source-DPOR algorithm for SMC/DPOR under sequential consistency. It is a simple, but efficient algorithm that was introduced in [1].

In [1], the authors present the algorithm in a general setting, without instantiating to any particular target programming language. Instantiating the algorithm to a target requires defining the function which maps a run to its corresponding happens-before order. The authors define a number of requirements on that function, for it to be compatible with the algorithm. Here, we will instantiate the algorithm for our assembly language under sequential consistency in a natural manner. The discussion will be kept informal. For the full detail, the reader is referred to [1].

Happens-Before Order

The happens-before order that we will use for our assembly language under SC was already alluded to in Example 4.1 about Mazurkiewicz traces above. It should capture the observable order between events in a run, and is therefore defined to be the program order, together with the order of pairs of memory accesses to the same location where at least one of them is a store. Formally, for an SC run \( \tau \) corresponding to a Shasha-Snir trace \( \pi = (E, po, co, rf) \), we define the relation \( \rightarrow_{\tau}^{hb} \subseteq E \times E \) as the partial order \( \rightarrow_{\tau}^{hb} = (po \cup co \cup rf \cup fr_\pi)^* \).

Algorithm Outline

The basic operation of the source-DPOR algorithm is the following: First the program is executed until termination using some arbitrary schedule. The resulting run \( \tau \) is examined, its happens-before order is computed, and conflicting events are identified. The algorithm is in particular identifying events that are in a race. Events \( e, e' \) of different threads in a run \( \tau \) are said to be in a race, in this context, when they are conflicting and concurrent, i.e., when \( e \rightarrow_{\tau}^{hb} e' \) in \( \tau \) and there is no other intervening event \( e'' \) in \( \tau \) such that \( e \rightarrow_{\tau}^{hb} e'' \rightarrow_{\tau}^{hb} e \).

For each pair of racing events \( e, e' \) corresponding to labels \( l \) and \( l' \) in the run \( \tau = \tau_0.l.\tau_1.l'.\tau_2 \), the algorithm needs to ensure that another run is explored, where the race is reversed, i.e., where \( e' \) precedes \( e \). In order to do this, the algorithm will associate the sub-run \( \tau_0 \) with a set of events, called the backtrack set. For each event \( e_b \) in the backtrack set with \( l_b = \text{label}(e_b) \), the run \( \tau_0.l_b \) and its continuations should be explored, where \( l_b \) is scheduled instead of the event labelled \( l \). Here, the algorithm would insert some event \( e_b \) into the backtrack set which is compatible with running \( e' \) before \( e \). The source-DPOR algorithm will pick any event \( e_b \) in \( \tau_1.l' \) which does not have a happens-before predecessor in \( \tau_1.l' \).

Furthermore, when the continuations of \( \tau_0.l_b \) are explored, it is necessary to remember that \( e \) should not be scheduled too early, such that no race is
reversed. For this purpose, a so called sleep set is used. When \( e_b \) is executed after \( \tau_0 \) instead of \( e \), the previous event \( e \) is entered into the sleep set. Events in the sleep set are temporarily blocked from being scheduled. The event \( e \) is not removed from the sleep set until some later event \( e_c \) (which could be the same as \( e_b \)) is executed such that \( e \) is in conflict with \( e_c \). This mechanism ensures that at least the race between \( e \) and \( e_c \) is reversed.

**Example 4.2 (Operation of Source-DPOR).** Let us consider again the program in Figure 4.1, and see how source-DPOR may operate when applied to this program.

Figure 4.3 illustrates the process. As explained above, we start by running the program once, using an arbitrary scheduling. Assume that the chosen order is \( \text{L0, M0, NO, N1} \). The first run \( \tau_0 \) is shown as the leftmost column in Figure 4.3. The happens-before order \( \rightarrow_{hb}^{\tau_0} \) is drawn in the figure as solid black (for program order) and red (for conflicts) arrows. At each position in the run two sets are shown, corresponding to the preceding prefix of the run. These are the sleep set to the left, and the backtrack set to the right. We see that for each event, that event is immediately inserted into the backtrack set of the sub-run leading up to the event. This is to represent that the event has now been explored as a continuation from this point in the run. In the first run, nothing has been inserted into the sleep sets. Studying the happens-before order, we can see that \( \tau_0 \) contains two races: \( \text{M0 to N1} \) and \( \text{L0 to NO} \). In order to reverse the race from \( \text{M0 to N1} \), we must find an alternative event that could be executed instead of \( \text{M0} \) in a run where \( \text{N1} \) precedes \( \text{M0} \). This alternative is picked from the postfix \( \text{NO.N1} \), and should not have a happens-before-predecessor. The only viable candidate is \( \text{NO} \). And so we insert \( \text{NO} \) into the backtrack set immediately above \( \text{M0} \) in \( \tau_0 \).
Similarly, to reverse the race from L0 to N0 a candidate must be chosen from M0. N0, N1. There are two candidates this time: M0 and N0. In the figure, we have arbitrarily chosen N0, and inserted it into the backtrack set immediately preceding L0.

Our next step will be to find a way to rewrite τ0 into a new different run, based on the events in our backtrack sets. We identify the latest position in the run where the backtrack set contains a new alternative continuation event which is not also in the sleep set. In τ0, this position is immediately after L0, where N0 could be executed instead of M0. In the next run τ1 (second to left-most column in Figure 4.3), we keep the prefix of the run leading up to this position, then we execute N0, and then we continue the run according to some arbitrary schedule. When we replace M0 by N0, we also insert M0 into the sleep set to prevent it from being executed too soon. Since M0 is in the sleep set after N0, we are not allowed to execute M0, and have to execute N1 instead. Since N1 is in conflict with M0, as both access y, we remove M0 from the sleep set after N1. Therefore, we are finally able to execute M0, and end the run. Notice how the combination of the backtrack set and the sleep set caused us to reverse the race from M0 to N1.

The run τ1 contains two races: L0 to N0 and N1 to M0. We proceed to selecting new alternative events into the backtrack sets in order to reverse these races. For the race from L0 to N0, the only alternative is to execute N0 instead of L0 as the first event of the run. However, N0 is already in the initial backtrack set, so no change is made. For the race from N1 to M0, the only alternative is to execute M0 after N0. However, we see that M0 is in the sleep set at this position and cannot be executed. So no change is made here either. Notice that the underlying reason that M0 is blocked at that position is that we have just reversed the race between M0 and N1, and that it would not be useful to reverse it again.

Again we find the latest position in the run where a different event may be picked and executed from the backtrack set. This time, we pick and execute N0 instead of L0 at the very beginning of the run. As we replace L0, we insert L0 into the sleep set at the beginning of τ2. When we execute N0, we remove L0 from the sleep set again, since N0 is in conflict with L0. Thereafter, the run continues according to some arbitrary schedule.

We discover a race from N1 to M0. When searching for an alternative event to enter into the backtrack set immediately above N1, we should pick a candidate from the sub-run L0.M0. Both L0 and M0 are eligible since neither has a happens-before-predecessor in L0.M0. In the example we pick L0.

Executing L0 in place of N1 leads us to the final run τ3, as shown on the very right of Figure 4.3. When we analyze this run, we find two races: N0 to L0 and M0 to N1. In both cases, we do not make any changes to the backtrack sets, since the candidate event is already found in the corresponding sleep set. Searching through the backtrack sets of τ3, we find that
there are no more events that can be executed instead of the ones in \(\tau_3\), except for the events which are already in the sleep sets. Therefore, the SMC procedure terminates. Comparing Figure 4.3 with Figure 4.2, we can see that the source-DPOR algorithm has successfully explored precisely one run corresponding to each Mazurkiewicz trace of the program.

**Sleep Set Blocking**

Source-DPOR guarantees that there is a one-to-one correspondence between Mazurkiewicz traces of a program and the runs which are completely explored by the source-DPOR algorithm. So no superfluous run is entirely explored. However, it is possible that exploration starts on a run which is later found to be superfluous and prematurely terminated. This happens when a run gets blocked because all remaining events are in the sleep set. Such a run is said to be **sleep set blocked**. Since sleep set blocked runs are superfluous, it is desirable for a SMC/DPOR algorithm to minimize the number of sleep set blocked runs.

**Example 4.3** (Sleep Set Blocking in Source-DPOR). Sleep set blocking can occur when analyzing the program of Example 4.2, if certain unfortunate choices are made about scheduling and about which events to include in the backtrack sets. Figure 4.4 shows how this can happen.

We first explore the run \(\tau_0\) in the same way as in the previous example. But this time, when addressing the race from \(L0\) to \(N0\), we elect to insert \(M0\) instead of \(N0\) into the backtrack set at the beginning of the run. Notice that both \(N0\) and \(M0\) are acceptable candidates, since both are in the sub-run \(M0.N0\), and neither has a happens-before predecessor in that sub-run.

The second run \(\tau_1\) is explored in the same way as before. This time, when we address the race from \(L0\) to \(N0\), we have no other choice than to
insert $N_0$ into the backtrack set at the beginning of the run, since $N_0$ is the only event between $L_0$ (exclusive) and $N_0$ (inclusive) in the run.

The exploration of $\tau_2$ and $\tau_3$ proceed in the same way as before. Since now both $L_0$ and $N_0$ have been explored as the first event in a run, both events are in the sleep set at the beginning of the run. Therefore only $M_0$ remains as a candidate for execution in the initial backtrack set. So the fifth run $\tau_4$ starts with $M_0$. Since $M_0$ conflicts with neither $L_0$ nor $N_0$, both remain in the sleep set after $M_0$ has been executed. There is no other event that can be executed, and so the run $\tau_4$ is sleep set blocked. In retrospect, we can see that $\tau_4$ is also superfluous, since all four Mazurkiewicz traces have already been explored in runs $\tau_0$ to $\tau_3$.

4.1.1 Source-DPOR Algorithm

Here we will recall the source-DPOR algorithm from [1]. It is given in Figure 4.5.

The recursive algorithm Source-DPOR takes one argument $\tau$ which is the current run, whose continuations should be explored. The initial call should be $\text{Source-DPOR}(\epsilon)$, to start with an empty run prefix. The algorithm keeps two global variables backtrack and sleep, each maps from prefixes of runs to the corresponding backtrack set and sleep set, as explained in previous sections.

On lines 1 and 2 we check if the run has terminated, or if there is still some event that can be executed. Here we use the function enabled($\sigma$), which returns the set of all events that are enabled in $\sigma$. Formally, for an SC state $\sigma = (\mathcal{M}, \mathcal{Q}, \mathcal{R})$, we define $\text{enabled}(\sigma)$ to contain precisely the events corresponding to the label $\mathcal{Q}(t)$ for each thread $t \in \mathcal{T}$ such that $\mathcal{Q}(t) \neq \bot_t$. If there are any such remaining events, an arbitrary enabled event $e_0$ which is not in the sleep set is selected as the first to explore, and added to the backtrack set of $\tau$. The loop on lines 4-21 iterates over those events that should be tried as a first event after $\tau$. Those events include the first selected event $e_0$, and any event which is added later for the purpose of reversing some race.

On line 4, we pick the next event $e$ to try after $\tau$. The run $\tau'$ is assigned the run $\tau$ followed by $e$. Lines 6-14 serve to detect races, and make certain that they are eventually reversed. Lines 15-20 perform the recursive exploration of the runs which are continuations of $\tau'$, and manages the sleep sets.

On line 6 we identify which previously executed events in $\tau$ are in a race with $e$ in $\tau'$. Recall that for a run $\tau'$ and two events $e'$ and $e$ with $\text{tid}(e) \neq \text{tid}(e')$, we say that $e'$ and $e$ race in $\tau'$ if $e' \rightarrow^{\text{rb}}_{\tau'} e$, and there is no other event $e''$ in between, such that $e' \rightarrow^{\text{rb}}_{\tau'} e'' \rightarrow^{\text{rb}}_{\tau'} e$. Then on lines 7 and 8 we split the run into a prefix and a subword of a postfix. Formally, for a run $\tau$ and an event $e'$ of $\tau$, let $\tau_0$ be the longest prefix of $\tau$ such that $e'$ is not an event of $\tau$. I.e., $\tau_0$ is the run up to but not including the label corresponding to $e'$. Then
global backtrack = λτ.∅;
global sleep = λτ.∅;

Source-DPOR(τ)
1: σ := τ(σ0,SC); // Current state
2: if (∃e0 ∈ (enabled(σ) \ sleep[τ])) {  
3:   backtrack[τ] := {e0}; // First candidate
4:   while (∃e ∈ (backtrack[τ] \ sleep[τ])) { // Pick event
5:     τ′ := τ.label(e);  
6:     // Detect races
7:     for (e′ s.t. e′ and e race in τ′) {  
8:       τe′ := pre(τ, e′);  
9:       τe := notdep(τ, e′).label(e);  
10:      I := initsτ,e′(τe);  
11:      if (I ∩ (backtrack[τe] \ sleep[τe]) = ∅) {  
12:        e″ := pick an element from I;  
13:        backtrack[τe] := backtrack[τe] \ {e″};  
14:      }  
15:     }  
16:   sleep[τ′] := sleep[τ]; // Propagate sleep set
17:   // Wake up conflicting sleepers
18:   for (e′ ∈ sleep[τ] s.t. e and e′ race in τ′.label(e′)) {
19:     sleep[τ′] := sleep[τ′] \ {e′};
20:   }
21:   // Continue exploration after τ′
22:   Source-DPOR(τ′);
23:   sleep[τ] := sleep[τ] ∪ {e}; // Did e
24: }

Figure 4.5. The source-DPOR algorithm of [1]. The initial call is Source-DPOR(ε).
\( \tau \) has the form \( \tau_0.\text{label}(e').\tau_1 \) for some sequence \( \tau_1 \) of labels. Now let \( \tau'_1 \) be the largest subword of \( \tau_1 \) such that the sequence \( \tau_2 = \tau_0.\text{label}(e').\tau'_1 \) is a run under SC, which contains no event \( e'' \neq e' \) such that \( e' \rightarrow^{\text{hb}} e'' \). I.e., \( \tau'_1 \) is \( \tau_1 \) with all events removed which succeed \( e' \) in happens-before order either directly or transitively. We now define \( \text{pre}(\tau, e') = \tau_0 \) and \( \text{notdep}(\tau, e') = \tau'_1 \). On line 9, we let \( \bar{I} \) be the set of events of \( \tau_e \) which do not have any \( \rightarrow^{\text{hb}} \)-predecessors in \( \tau_e \). The purpose of all this is to have in the set \( \bar{I} \) all the events that could be executed in place of \( e' \) after \( \tau_e' \), in order to reverse the race between \( e' \) and \( e \). If no such event is already represented in the backtrack set or sleep set of \( \tau_e' \), then we add some element from \( \bar{I} \) to the backtrack set of \( \tau_e' \) on line 12.

On lines 15-18 we propagate the current sleep set to the next step in the exploration of the current run. All events that are currently sleeping (i.e., events in \( \text{sleep}[\tau] \)) carry over to \( \text{sleep}[\tau'] \), except for those which are in conflict with \( e \).

Finally, on line 19 we make the recursive call which continues the exploration of \( \tau' \), and then add \( e \) to the sleep set to ensure that the same event is not executed again from \( \tau \), before some other conflicting event has been executed.

### 4.2 Chronological Traces for TSO and PSO

We have seen how SMC/DPOR can be used to analyze programs running under sequential consistency. In this section, we will see how chronological traces allow us to extend those techniques to the relaxed memory models TSO and PSO. This section is a brief introduction to the main results of Paper III.

SMC/DPOR under SC uses a happens-before order and its corresponding Mazurkiewicz traces. They have two purposes: to define equivalence classes of runs corresponding directly to observable behaviors, and to enable us to identify races that should be reversed. For relaxed memory models, it is common to use Shasha-Snir traces to succinctly express the observable behavior of a run. Similar to Mazurkiewicz traces for SC, Shasha-Snir traces can be used to define natural equivalence classes for runs under relaxed memory. Both Mazurkiewicz traces and Shasha-Snir traces are precisely defined by the events of a run and their program order, coherence order and read from relation. This means that under SC, the Shasha-Snir traces of a program coincide precisely with the Mazurkiewicz traces. It makes sense therefore to consider Shasha-Snir traces as appropriate equivalence classes when extending DPOR to TSO and PSO. However, it is not possible to directly apply existing dynamic partial order reduction techniques to Shasha-Snir traces under relaxed memory. This is because Shasha-Snir traces in general are not partial orders for programs under relaxed memory. As we have seen in Chapter 3, Shasha-Snir traces contain cycles for precisely those runs that exhibit observable non-SC behavior.
In Figure 4.6, we recall a run under TSO of the Dekker idiom, and its corresponding cyclic Shasha-Snir trace.

The first thing that is necessary in order to be able to apply dynamic partial order reduction to these traces is to change the representation such that it becomes a partial order. While doing so, we still need to ensure that the happens-before order captures both the event orderings which are required to be enforced (e.g. $R \rightarrow R$ order under TSO) and the races which need to be reversed in order to explore all different behaviors. This can be achieved for TSO and PSO by changing the Shasha-Snir trace representation by introducing explicit events for memory updates. The new trace representation of the run in Figure 4.6 is shown in Figure 4.7. As can be seen in the figure, program order relations remain as before. In addition, edges are introduced from stores to their corresponding updates. These edges are marked $su$ in the figure, and they are considered irreversible, similar to program order edges. Edges representing races between memory accesses remain the same as before, except that the update events have now taken the place of the store events. Therefore, in Figure 4.7, we now have two reversible race edges, marked in red. It is easy to see that every trace constructed in this manner from a TSO or PSO run is acyclic. This follows from the fact that each edge in such a trace goes from an event $e$ to an event $e'$ such that $e$ appears before $e'$ in the corresponding TSO or PSO run.

The trace definition we have now formulated can be used with off the shelf SMC/DPOR techniques (e.g. the source-DPOR algorithm described above), which will then explore at least one run per Shasha-Snir trace of the target...
program. However, the representation is still flawed, in the sense that it is not canonical. The same Shasha-Snir trace may correspond to multiple traces in this representation. Hence if we apply SMC/DPOR to this representation, it will waste time by exploring more than one run per observable behavior. To see this, assume that the memory model is TSO, and consider the small program shown in Figure 4.8. The six traces of the program, by our current representation, are shown in Figure 4.9. Each trace has two update events, both targeting the memory location $x$, and one load event, also targeting $x$. These three events can occur in any order, and since each pair constitutes a race, there are six different traces. However, if we consider the Shasha-Snir traces of the same program, there are only three of them. They are shown in Figure 4.10. The dashed outlines in Figure 4.9 indicate how multiple traces in our new representation correspond to the same Shasha-Snir trace. Comparing trace (b) with trace (c), we see that in both cases, the store $L_0$ precedes $M_0$ in coherence order, and in both cases the load $L_1$ reads the value written by $L_0$. The only difference between the traces is that in (b) $L_1$ reads the value from memory, while in (c) it reads the same value by buffer-forwarding. The differences between (d), (e) and (f) are similar.

We will now change our trace representation by adding two new rules for how edges are inserted. The result will be a trace representation called *chrono-logical traces*. They have the properties that off the shelf SMC/DPOR techniques can be directly applied to them, and furthermore they induce precisely the same equivalence classes over runs as Shasha-Snir traces do. Our comparison of the traces (b) and (c) in Figure 4.9 leads us to the first realization. When a store $e_w$ precedes a load $e_r$ to the same memory location in program order, it is not necessary to keep track of the order between the load $e_r$ and the update event corresponding to $e_w$. This is because changing that order can only ever cause loads to change between reading from memory and reading the same value from the thread’s store buffer. Therefore, in chronological traces, there are never any direct edges between such updates and loads. This is the first of the two rules.

Now consider the traces (e) and (f) in Figure 4.9. They differ only in the order between the load $L_1$ and the update corresponding to $M_0$. However, this order is irrelevant, since in both cases the load $L_1$ has to read its value from the pending store $L_0$ in the store buffer of $P$. Intuitively, the update of $M_0$ is

$$x = 0$$

thread $P$: thread $Q$:
$L_0$: $x := 1$; $M_0$: $x := 2$;
$L_1$: $r_0 := x$;

Figure 4.8. A small program exhibiting buffer forwarding under TSO and PSO.
Figure 4.9. The program in Figure 4.8 has six traces (a-f) according to the suggested trace representation. These six traces correspond to only three Shasha-Snir traces. The dashed outlines indicate which traces correspond to the same Shasha-Snir trace.

Figure 4.10. The three Shasha-Snir traces of the program in Figure 4.8.
hidden from \textbf{L1} by the pending store \textbf{L0}. This leads us to the second rule, which intuitively says that loads are not ordered with such “hidden” updates: Assume that $e_w$ is a store to some memory location $x$, which program order-precedes a load $e_r$ to $x$. Assume that $e'_w$ is some store to $x$ such that the update of $e'_w$ is ordered before the update of $e_w$ in the chronological trace. Then there is no direct edge between $e_r$ and the update of $e'_w$ in the chronological trace.

When these two rules are applied to the traces in Figure 4.9, the six traces are reduced into the three chronological traces shown in Figure 4.11. The chronological traces correspond directly to the Shasha-Snir traces shown in Figure 4.10.

In Paper III it is proven that chronological traces induce exactly the same equivalence classes over runs as Shasha-Snir traces do, while at the same time satisfying the requirements on happens-before orders imposed by source-DPOR. Furthermore the paper shows how the chronological trace can be efficiently computed on the fly while the run is being explored. The technique is implemented in the tool Nidhugg [34]. Experimental results are given in Paper III.

\section*{4.3 RSMC for POWER}

This section concerns my contributions made in Paper IV. Here we will see how stateless model checking can be applied to the more relaxed memory model of POWER.
While chronological traces allow us to apply classical DPOR algorithms to TSO and PSO, it turns out that they are not as readily adapted to POWER. This is because the operational models of POWER (such as e.g., [46]) make use of a large number of internal transitions, which interact in complex ways. In particular the phenomenon of non-atomic stores makes it difficult to construct a formulation of chronological traces for POWER which simultaneously captures the dependencies between events and maintains a one-to-one correspondence with Shasha-Snir traces. For this reason, we introduce a different technique for applying SMC to POWER: the RSMC algorithm.

In this section we will first see how the axiomatic model of POWER, based on the work by Alglave et al. in [14] and recalled in Section 3.3, can be transformed into an operational semantics of a certain form. Then we will explore the RSMC (Relaxed Stateless Model Checking) algorithm which extends stateless model checking to work for relaxed memory models expressed in that certain operational form. The RSMC algorithm is based on, but distinct from, stateless model checking with dynamic partial order reduction.

### 4.3.1 Operational POWER

As we saw in Chapter 3, axiomatic memory models are predicates which decide for a complete trace, whether it is allowed or not. This makes axiomatic memory models unsuitable for stateless model checking, since SMC is based on exploring runs in a stepwise fashion. We need therefore to produce a new model for POWER which allows us to execute events one at a time and to know at the time when the event is executed how it interacts with previously executed events. Furthermore, when we are exploring a run we want to be certain that the run is an actual run which is allowed by the POWER memory model. If we explore runs until termination only to then find out that the run is not an actual POWER run, then the effort has been wasted, and the performance of the analysis tool will suffer. For this reason we introduce a scheme, which allows us to explore runs in an operational fashion, while using the axiomatic model as a black box to ensure that we are producing only allowed runs.

Our operational model will work by the following main principles: A program instruction is first *fetched* as one operational step, then *committed* as another operational step. There are no additional operational steps corresponding to buffers updating to memory or similar. Fetching of instructions occurs in program order. When a memory access instruction is committed, it is equipped with a parameter which determines the effect of the instruction with respect to the other memory accesses in the run. For a store instruction, the parameter is a natural number which determines its position in the coherence order over stores to that same memory location. For a load instruction, the parameter is an identifier for a previously committed store. The load is then interpreted as reading the value written by the parameter store. The operational state consists
Figure 4.12. Left: The small program LB+ctrl. Right: An operational run of LB+ctrl under POWER.

Figure 4.13. Two states occurring in the run in Figure 4.12 (right). Left: The state immediately after line 6 in the run. Notice that \textbf{L0} and \textbf{M1} are fetched but not committed. Right: The state immediately after line 12 in the run.

of a partial Shasha-Snir trace built up of all hitherto committed instructions, together with extra events corresponding to all instructions which have been fetched but not committed. Notice how the parameters of stores and loads precisely determine the co and rf relations of the Shasha-Snir trace as we commit the memory accesses. Next we give an example of a run under our operational semantics, to give the gist of how it operates, before explaining the details of how it works.

\textbf{Example 4.4} (Operational Run under POWER). Figure 4.12 (left) shows the small program LB+ctrl. Figure 4.12 (right) shows an operational run of the program. On lines 1-4 the instructions \textbf{L0}, \textbf{L1}, \textbf{M0} and \textbf{M1} are fetched. Notice that \textbf{M2} and \textbf{M3} cannot be fetched at this point, since the conditional
branch \( M_1 \) has not yet been resolved, and so it is not known which of \( M_2 \) and \( M_3 \) succeeds \( M_1 \) in program order.

On line 5 we commit the store to \( y \). Both instructions \( L_0 \) and \( L_1 \) are fetched for the thread \( P \). Here we elect to commit them out of order. When we commit the store \( L_1 \), we equip it with a parameter "[0]", indicating its position in the coherence order over stores to \( y \). Since no other stores to \( y \) have been committed, \( L_1 \) gets the parameter 0, indicating that it is positioned immediately after the initializer store to \( y \) in the coherence order.

On line 6 we commit the load from \( y \). Here, we have a choice of parameter: Either we could read the initial value of \( y \), or we could read the value written by \( L_1 \). In this example, we choose to read the value 1 from \( L_1 \).

We can then on line 7 commit the conditional branch. Since the register \( r1 \) holds the value 1, control flow will continue at \( M_2 \). On lines 8-9, we fetch the remaining instructions \( M_2 \) and \( M_3 \). The no-op \( M_3 \) is immediately committed on line 10.

On line 11 we store 1 to \( x \). Again we give the store the parameter 0, since it is the first store to \( x \) after the initializer store.

Finally, on line 12 we commit the load \( L_0 \) from \( x \). Here we could choose to read the initial value of \( x \), or to read the value written by \( M_2 \). We choose the latter, and give \( L_0 \) the parameter \( M_2 \).

Figure 4.13 shows two of the operational states occurring in the run. On the right we see the final state of the run, where all instructions have been fetched and committed. The final state is the same as the Shasha-Snir trace corresponding to the entire run. On the left in Figure 4.13 we see the earlier state which occurs immediately after the load \( M_0 \) has been committed on line 6. The instructions \( L_0 \) and \( M_1 \) are shown in gray, to indicate that they are fetched in the state, but not committed. The instructions \( M_2 \) and \( M_3 \) have not been fetched, and are therefore not part of the state at all. We see that the parameter choice \( L_1 \) for \( M_0 \) induces a read from edge from \( L_1 \) to \( M_0 \).

In order to ensure that our operational model produces only runs that are allowed under POWER, we use the axiomatic model of [14] as a blackbox backend. Every time a memory access event is about to be committed, the operational state is updated by inserting the event into the Shasha-Snir trace and updating the coherence and read from relations. Then the axiomatic model is applied to the Shasha-Snir trace, to see whether it is allowed under POWER. If it is, then the operational model allows the memory access event to be committed with the chosen parameter. Otherwise, the transition is blocked, and the event will have to be committed with a different choice of parameter.

In Example 4.4 we committed the store \( L_1 \) before the load \( L_0 \), contradicting the program order between the instructions. That reordering was necessary to realize the Shasha-Snir cycle in the final state. However, not all pairs of events may be committed out of order by the operational model. For exam-
ple, it would not make sense to allow the store $M_2$ to commit before the load $M_0$. This is because it is not yet known whether the conditional branch $M_1$ will be taken, and whether $M_2$ will appear in the run at all, until the load $M_0$ has been committed. In our operational model, we will introduce a partial order $\text{cb}$ (“commit-before”) which orders precisely those events that must be committed in program order. An operational transition is allowed to commit an event $e$ only if for all events $e'$ such that $(e', e) \in \text{cb}$ it holds that $e'$ is already committed in the state. We define $\text{cb}$ to contain pairs of events $(e', e)$ where $e$ should not be committed before $e'$ because it is not yet known whether $e$ will be part of the run, or because the semantic effect of $e$ cannot yet be determined (e.g. because it is not known which memory address it is going to access). For POWER, we give the following definition.

$$\text{cb} = \text{addr} \cup \text{data} \cup \text{ctrl} \cup (\text{addr}; \text{po}) \cup \text{po-loc} \cup \text{sync} \cup \text{lwsync} \cup \text{rf}$$

The complete description of the operational POWER model can be found in Paper IV. In the same paper, we formally state and prove two important properties of the operational model: It is precise, in the sense that it admits precisely those complete runs which correspond to Shasha-Snir traces allowed by the axiomatic POWER model of [14]. Furthermore, the operational model will never run into a deadlock before the run reaches termination.

4.3.2 RSMC

The purpose of the RSMC algorithm for POWER is to explore at least one run, in the operational model given above, per allowed Shasha-Snir trace of the target program. Its operation is based on similar ideas as Source-DPOR (as described in Section 4.1), but differs in a few ways.

First, SMC/DPOR is based on the idea that equivalence classes of runs are characterized by partial orders, and enumerates the equivalence classes by gradually rewriting the partial orders. In RSMC, the $\text{cb}$ order plays a similar role to the happens-before order of SMC/DPOR. However, in RSMC there is no direct correspondence between $\text{cb}$ orders and equivalence classes of runs that need to be explored. Instead RSMC is mainly concerned with exploring different parameter choices for events. The parameter choices entirely determine the $\text{cb}$ order, but the converse is not true as multiple parameter choices may induce the same $\text{cb}$ order.

Another difference is in how RSMC handles races. In most cases, when RSMC detects a race in a run, it reacts by ensuring that another run is explored where the racing events are committed with different parameter choices. However, in the case of races from earlier loads to later stores, a more complicated mechanism is employed. Since RSMC needs to explore all observable behaviors of the program, it must ensure that a run is explored where the load
reads the value of that store. Since a load can only read the values written by stores which are committed before the load, this case cannot be handled by simply changing the parameter choice for the load. Instead it is necessary to change the order in which events are committed in the run. In Source-DPOR such reordering would be achieved by adding some event to the backtrack set, such that a different run is explored where the load is delayed, and the other event executed in its stead. In Source-DPOR under SC this would be sufficient to guarantee that a run is explored where the load reads the value written by the store. In RSMC under POWER, it is necessary to control more strictly the run that reverses the race. For this reason the backtrack sets in RSMC keep not only a single event, but an entire sub-run of events with fixed parameter choices leading up to the store event and the load event reading from the store. This mechanism is reminiscent of the wake-up trees used in Optimal-DPOR [1], but in the case of RSMC the mechanism is necessary to ensure soundness, whereas in Optimal-DPOR wake-up trees are used to achieve optimality.

The RSMC algorithm is described in detail in Paper IV. Here, we summarize its basic operation. The general outline of the RSMC algorithm is the same as Source-DPOR: We first explore one run according to an arbitrary schedule, and with arbitrary parameter choices. Then we identify the positions in the run where different parameters could be used, or where the schedule should be changed. We perform such a change at the latest changeable position in the run, thereby producing a new run. This is repeated until there are no more changes to be done.

Each time we commit a memory access event, a parameter must be chosen by the operational model. Recall that the operational model will use the axiomatic memory model as a blackbox to check whether the parameter is allowed. In RSMC, this check is performed for all possible parameters of the committed event. One such allowed parameter is arbitrarily chosen for the event in the first run. The remaining allowed parameters are kept in the backtrack set, so that they can be explored later.

The technique of trying different allowed parameters works well for enumerating all possible coherence orders between stores, and for enumerating all possibilities for a load to read the value written by previously committed stores. However, in the cases where a load races with a store which is committed later, another technique is required. When a store is committed in RSMC, we identify the previously committed loads to the same memory location. If such a load is to be able to read the value of the current store in an alternative run, then the load must be committed later than the store in that run. That can only happen if the load is not ordered before the store in the cb order. For those earlier loads to the same memory location, which are not cb-before the current store, we say that the load and store constitute a \( R \rightarrow W \) race. When a \( R \rightarrow W \) race is detected between some load \( e_r \) and store \( e_w \) in a run \( \tau \), the task of RSMC is to ensure that later a different run \( \tau' \) is explored where \( e_w \) is committed before \( e_r \), and \( e_r \) reads the value of \( e_w \). To ensure that \( e_w \) can be
executed in $\tau'$, RSMC must ensure that $\tau'$ contains all the events which pre-
cede $e_w$ in the cb order in $\tau$. Therefore, a sub-run is added to the backtrack
set of $\tau$ at the position where $e_r$ is committed. The sub-run consists of three
parts: all events in $\tau$ which cb-precede $e_w$ using the same parameters as in $\tau$,
the store $e_w$ itself, and the load $e_r$ with $e_w$ as parameter. In the same way as
alternative parameters are inserted into the backtrack set and explored later,
the sub-runs in the backtrack sets are also explored later and cause new runs
with distinct behaviors to be produced.

In Paper IV we describe the RSMC algorithm in more detail. We also state
and prove the following property of RSMC for POWER. The RSMC algo-
rithm for POWER will explore at least one run corresponding to each allowed
Shasha-Snir trace of the input program under POWER. This property tells us
that the RSMC algorithm is sound for POWER. We also show that the RSMC
algorithm will not explore more than one complete run for any Shasha-Snir
trace. However, the algorithm will occasionally explore superfluous incom-
plete runs, which are aborted before they complete. The technique is imple-
mented in the tool Nidhugg [34], and experimentally evaluated in Paper IV.
The experiments show that the technique can be successfully applied to non-
trivial parallel benchmarks with performance which is comparable to other ex-
isting techniques.
Fence Synthesis

We have seen in previous chapters how memory ordering relaxations may cause errors in programs, and we have seen how various kinds of fence instructions may be used to enforce sufficient order to maintain correctness. However, correctly inserting fences in a program requires detailed knowledge of the underlying system’s memory model, and even with that knowledge it is difficult [41, 43, 38]. A naive solution would be to insert fences between all instructions in all threads. However, fences carry performance penalties since their operation requires obstructing the optimizations of the system that give rise to the memory access reorderings in the first place [17, 24, 50]. A different approach is to use various synchronization primitives provided by concurrency libraries, such as mutual exclusion locks, condition variables etc. As discussed in Section 3.1.4, many systems guarantee that if a program uses synchronization primitives to eliminate all data races, then no memory relaxations will be observable. However, such synchronization primitives, and mutual exclusion locks in particular, incur their own performance penalties, as they require threads to wait for each other. To improve performance and scalability, it is therefore sometimes desirable to instead employ various lock-free algorithms [24]. This again requires the programmer to worry about relaxed memory, and inserting sufficient fences to maintain correctness.

So, depending on the program, inserting fences may be necessary for correctness. Since fences incur performance penalties, we want to insert as few of them as possible. Furthermore, some relaxed memory models, such as PSO and POWER, provide multiple different kinds of fences. The different kinds may be associated with different performance penalties, since they employ different techniques for enforcing order, and disable different hardware optimizations and different kinds of memory access ordering relaxations.

To make the difficult job of fence insertion easier for the programmer, several techniques for automatic fence synthesis have been proposed. In [30], Kuperstein et al. suggest a technique for finding minimal sets of fences which are sufficient for the program to satisfy a safety specification under the given
memory model. The technique operates by computing cuts in the finite state transition system of the program. In [5], we propose a verification technique for programs under the TSO memory model, together with a counter example guided fence insertion algorithm. The algorithm runs the verification engine multiple times and uses the generated counter examples (error runs) to infer minimal fence sets, sufficient to maintain a safety specification under TSO. In [37], Liu et al. introduce a counter example guided fence insertion algorithm using a dynamic analysis back end. In [35], Linden and Wolper introduce a fence insertion technique for guaranteeing safety specifications under the PSO memory model. It works by first analyzing the program under TSO, and inserting full memory fences to maintain the necessary $W \rightarrow R$ memory access orderings. When the necessary full fences are inserted, the augmented program is then analyzed under PSO, and the store fences which are needed to maintain the necessary $W \rightarrow W$ memory access orderings are inferred. In [40], Meshman et al. propose an algorithm which combines fence inference with abstraction refinement for abstract interpretation for infinite state programs. They apply their technique to the TSO and PSO memory models.

All of the above works address the problem of inserting fences in a program in order to make it satisfy a given safety specification. A different approach is to insert fences in a program in order to make it satisfy the robustness criterion. The robustness criterion is based on the notion of Shasha-Snir traces [49]. A program run under a given memory model has the same observable behavior as some run under SC, if and only if its Shasha-Snir trace is acyclic. A program is said to be robust under a given memory model if all of its runs under the memory model correspond to acyclic Shasha-Snir traces. In [18], Bouajjani et al. show how to automatically insert fences to guarantee robustness under TSO. The technique uses reachability analysis over an instrumented version of the program code. In [13], Alglave et al. propose a technique using abstract interpretation in order to efficiently find over approximate fence sets which are sufficient for guaranteeing robustness under various memory models.

The robustness approach has the advantage that the programmer does not need to provide a specification for their program, and can automatically infer fences anyway. The safety specification approach requires the programmer to make an effort in specifying which behaviors their program should have, but in return the automatic fence insertion techniques based on safety specifications are often able to infer smaller, hence better performing, fence sets.

In this chapter, we will explore one technique for automatic fence insertion, using the safety specification approach. The algorithm is essentially based on the one presented by Liu et al. in [37]. It is counter example guided, and therefore requires an underlying analysis engine that can check satisfaction of safety specifications under the given memory model, and provide counter examples (error runs) in case the program does not satisfy its specification. Provided such an analysis engine, the fence insertion algorithm is otherwise applicable to any memory model. A main feature of the algorithm is that it
works in the presence of multiple different kinds of fences. Here we extend
the technique with a cost function which assigns different cost metrics to dif-
ferent fences. The algorithm then finds all optimal sets of fences for the given
program, specification and cost function. By optimal, we mean that the fence
set(i) is sufficient for the program to satisfy its specification, and (ii) has the
least total fence cost among all fence sets that are sufficient for the program to
satisfy its specification. A formal definition of optimality is given later in this
chapter.

Papers I and II contain an older fence insertion algorithm than the one pre-
presented in [37]. Both algorithms work by repeatedly running a reachability
analysis on the program, and deriving information from the resulting error runs
which is used to infer the necessary fence placements. The algorithms differ
in how this information is used. In particular, the older algorithm explores
different possible fence placements in a recursive, tree-shaped manner. This
means that information which is discovered in one recursive branch, is un-
available when fence placements are inferred in another recursive branch. The
newer algorithm avoids this problem by accumulating all information in one
place (the set $$\mathcal{R} \subseteq$$ below). This reuse of information may reduce the number
of calls to the expensive reachability analysis, and thereby improve the total
performance of the fence insertion procedure.

5.1 A Fence Insertion Algorithm
This section presents a variation of the fence insertion algorithm introduced by
Liu et al. in [37].

Up to now we have used the expression “inserting fences into a program”
vaguely. Let us formalize the notion. We will use the term fence constraint
(sometimes just fence for short) to denote a pair $$(f, l)$$, where $$f$$ is a kind of
fence instruction (fence, sfence, sync, ...) and $$l$$ is a description of how
to insert that fence into a program. In general $$l$$ can be any description suit-
able to the program and programming language currently considered. Here,
for simplicity, we will assume that $$l$$ is a label, and that inserting $$(f, l)$$ into a
program, amounts to inserting $$f$$ in the program code immediately following
the instruction labelled $$l$$. For a program $$P$$ and a fence constraint $$(f, l)$$, we
let $$P \oplus (f, l)$$ denote the program $$P$$ with $$(f, l)$$ inserted in that manner. Next,
we will define what it means to insert multiple fences into the same program.
In order for this operation to be well-defined, even when multiple fence con-
straints $$(f, l)$$ and $$(f', l)$$ should be inserted to the same position $$l$$, we assume
that some total order $$\prec$$ is given over all possible fence constraints. We can
then define $$P \oplus F$$ for a set $$F$$ of fence constraints recursively: $$P \oplus \emptyset = P$$
and $$P \oplus F = P' \oplus F'$$ for $$F \neq \emptyset$$ where $$(f, l)$$ is the $$\prec$$-least fence constraint
in $$F$$ and $$P' = P \oplus (f, l)$$ and $$F' = F \setminus \{(f, l)\}$$.

81
Parameters to the Fence Insertion Algorithm.
The fence insertion algorithm takes as arguments a program $\mathcal{P}$, a safety specification $\phi$, and a fence constraint cost function $\kappa$. Furthermore, it assumes that a reachability analysis engine $\text{Reachable}$ for the given memory model is provided, as well as a witness analysis function $\text{Analyze-Witness}$.

The fence constraint cost function $\kappa$ is a function from fence constraints to positive integers, indicating some cost metric of each fence constraint. We will overload $\kappa$ to sets of fence constraints in the natural way: $\kappa(F) = \sum_{f \in F} \kappa(f)$.

The reachability analysis $\text{Reachable}$ should be a function taking as arguments a program $\mathcal{P}$ and a safety specification $\phi$, which returns $\bot$ if $\mathcal{P}$ satisfies the specification $\phi$ under the given memory model, and otherwise returns a run $\tau$ under the given memory model which leads from the initial state to a state which is forbidden by $\phi$. The function $\text{Reachable}$ could be instantiated with any of the reachability analysis techniques described in papers I, II, III or IV.

The purpose of the witness analysis function is to discover which new fences need to be inserted into the program in order to prevent an error. The witness analysis is called with the program and with an error run as arguments. It computes a set of fences that can make the error run impossible, and these are used by the fence insertion algorithm to guide the search for optimal fence sets. In Section 5.2 we detail the requirements on the witness analysis function, and provide pointers for how one can be implemented.

Output from the Fence Insertion Algorithm.
The fence insertion algorithm returns all optimal fence constraint sets with respect to the program $\mathcal{P}$, specification $\phi$ and cost function $\kappa$. We will now formalize what this means.

\textbf{Definition 5.1} (Sound Fence Set). A fence constraint set $F$ is \textit{sound} with respect to the program $\mathcal{P}$ and specification $\phi$, if the fenced program $\mathcal{P} \oplus F$ satisfies $\phi$.

\textbf{Definition 5.2} (Optimal Fence Set). A fence constraint set $F$ is \textit{optimal} with respect to the program $\mathcal{P}$, specification $\phi$ and cost function $\kappa$, if $F$ is sound w.r.t. $\mathcal{P}$ and $\phi$, and there is no fence constraint set $F'$ with $\kappa(F') < \kappa(F)$ which is also sound w.r.t. $\mathcal{P}$ and $\phi$.

Hitting Sets
The last notion we need to introduce before explaining the workings of the fence insertion algorithm is that of hitting sets.
Definition 5.3 (Hitting Set). For a set of sets \( S = \{S_0, S_1, \cdots, S_n\} \), we say that a set \( T \) is a hitting set of \( S \), if for each \( 0 \leq i \leq n \) we have \( T \cap S_i \neq \emptyset \).

We will assume that a function \( \text{Hits}(S, \kappa) \) is given, which computes all hitting sets for \( S \), which are cheapest w.r.t. \( \kappa \). I.e., for all \( S \) and \( \kappa \) we have \( \text{Hits}(S, \kappa) \not= \emptyset \) and for each \( T \in \text{Hits}(S, \kappa) \), we have that \( T \) is a hitting set of \( S \), and that there is no hitting set \( T' \) of \( S \) such that \( \kappa(T') < \kappa(T) \).

Computing cheapest hitting sets is an NP-hard problem (Theorem 5.1). It can be solved by e.g. integer programming or constraint programming. In the implementation in Memorax, it is solved by a straight-forward search algorithm.

Theorem 5.1. The problem of computing a cheapest hitting set for a set \( S \) of sets w.r.t. a cost function \( \kappa \) is NP-hard.

Proof of Theorem 5.1. We can reduce the NP-complete decision problem of Node Cover [29] to the problem of computing a cheapest hitting set. The Node Cover decision problem is stated as follows: Given a graph \( \langle V, E \rangle \) and an integer \( k \), decide if a set \( U \subseteq V \) exists such that \( |U| \leq k \) and for each edge \((u, v) \in E \) it holds that either \( u \in U \) or \( v \in U \). We encode this into the problem of computing a cheapest hitting set by assigning \( \kappa = \lambda v.1 \) and \( S = \{\{u, v\}|(u, v) \in E\} \). The answer to the decision problem is given by comparing the cardinality of the cheapest hitting set \( T \) of \( S \) w.r.t. \( \kappa \) with \( k \).

The Fence Insertion Algorithm

We are now ready to start examining the fence insertion algorithm in Figure 5.1. The algorithm uses as data structures two sets \( \text{opt} \) and \( \text{req} \). Both are sets of sets of fence constraints, but they have very different interpretation.

The set \( \text{opt} \) is where we collect optimal fence constraint sets as we discover them. When a fence constraint set is added to \( \text{opt} \), it will never be removed from the set. The invariant for \( \text{opt} \) which holds after line 1, is that every fence constraint set \( F \) in \( \text{opt} \) is optimal w.r.t. \( \mathcal{P}, \phi \) and \( \kappa \).

The set \( \text{req} \) is used to keep track of which fence constraints are necessary for soundness of \( \mathcal{P} \). Each time we find an error run, we will add an inner set of fence constraints to \( \text{req} \), which contains all the alternative ways to avoid that error. Hence in order to avoid all errors that have been found thus far, it is necessary to have at least one fence constraint from each inner set in \( \text{req} \). The invariant for \( \text{req} \) which holds after line 2, is that every possible fence constraint set \( F \) which is sound w.r.t. \( \mathcal{P} \) and \( \phi \) is a hitting set of \( \text{req} \). Notice that the invariant holds immediately after initialization, since every set is a hitting set of the empty set.

\footnote{Here we deviate slightly from the usual definition of hitting sets (see e.g., [29]), which requires a hitting set to have \textit{exactly one} element in common with each set \( S_i \).}
Figure 5.1. The fence insertion algorithm.

The main loop on lines 3-11 runs until all optimal fence constraint sets are in $\text{opt}$. In each iteration, a new candidate fence constraint set $F$ is chosen on line 3. It is chosen to be one of the cheapest hitting sets of $\text{req}$, and is also required to not already have been inserted into $\text{opt}$. If there is no such candidate, then it must be the case that all optimal fence constraint sets have already been discovered, since all remaining sound sets (i.e., hitting sets of $\text{req}$) are more expensive than the optimal ones.

On line 4, we call the reachability engine, in order to test if the candidate $F$ is sound. If the reachability engine finds that the fenced program $\mathcal{P} \oplus F$ satisfies its specification $\phi$, and returns $\bot$, then we know that $F$ is sound w.r.t. $\mathcal{P}$ and $\phi$. Furthermore, we even know that $F$ is optimal w.r.t $\mathcal{P}$, $\phi$ and $\kappa$. To see this, remember that by the invariant every sound fence set is a hitting set of $\text{req}$, and that $F$ was chosen as one of the cheapest hitting sets of $\text{req}$ w.r.t. $\kappa$. Therefore, we can insert $F$ into $\text{opt}$ on line 6.

If the fenced program is found to violate its specification, then the reachability engine returns a witness run $\tau$ which leads from the initial state to an error state. Then, on line 8, we analyze $\tau$ with the witness analysis procedure. In Section 5.2, we describe in detail the requirements on a witness analysis procedure for it to work with the algorithm in Figure 5.1, and provide hints for how it can be implemented. The procedure returns a set $C$ of fence constraints. Typically, the set $C$ is computed such that it contains all alternative fence constraints which can make the error run $\tau$ irreproducible. Since any sound fence set must prevent the error run $\tau$, any sound fence set must then
have a non-empty intersection with \( C \). Therefore we add the inner set \( C \) to \( \text{req} \) on line 9. By doing so, we strengthen the restrictions on all future candidate fence constraint sets that will be chosen by the algorithm on line 3. Effectively, during each iteration of the main loop, where the reachability engine finds an error, we increase our knowledge about the behaviors of the program, and we collect this knowledge in \( \text{req} \) and use it to guide our search for optimal fence sets.

One special case merits additional concern. Assume that the witness run \( \tau \) does not exhibit any non-SC behaviors. Then \( \mathcal{P} \) is erroneous even under the SC memory model, and cannot be repaired by inserting fences. If \( \tau \) has no non-SC behaviors, then no fence constraints can prevent it, so \( C \) is empty. Then the empty set is added to \( \text{req} \) on line 9. In general, if a set \( S \) contains the empty set, then \( S \) has no hitting sets. Therefore the loop condition will fail in the next iteration of the main loop, and the algorithm will terminate, returning the current set \( \text{opt} \). At this point, the set \( \text{opt} \) will be empty. This is because any candidate fence set \( F \) is checked for soundness before being inserted into \( \text{opt} \), and any fence set \( F \) will fail the soundness check, since it cannot prevent the error run \( \tau \). So we can see that in this case, the algorithm correctly terminates and returns the set of all optimal fence sets for \( \mathcal{P} \), namely the set of no solutions.

5.2 Witness Analysis

The witness analysis procedure \textbf{Analyze-Witness} \((\mathcal{P}, \tau)\) needs to be instantiated to fit the given memory model. It takes as arguments a program \( \mathcal{P} \), possibly with some fences already inserted, and an error run which leads from the initial state in \( \mathcal{P} \) to an error state (according to some specification \( \phi \)). The witness analysis procedure returns a set \( C \) of fence constraints. The user has significant freedom when instantiating \textbf{Analyze-Witness}, but needs to satisfy these two post conditions:

1. Every fence constraint set \( F \) which is sound w.r.t. \( \mathcal{P} \) and \( \phi \), must have a non-empty intersection with \( C \), and
2. \( C \) contains no fence constraint which is already present in \( \mathcal{P} \).

The first of these requirements may seem difficult to satisfy, but is actually not.

One naive, but correct, way to implement \textbf{Analyze-Witness} is to simply return the set of all fence constraints which are not already present in \( \mathcal{P} \). This implementation trivially satisfies both requirements above. However, this witness analysis instantiation will not yield an efficient fence insertion procedure. Because the witness analysis does not make use of any knowledge about the program or the error run \( \tau \), the fence insertion procedure will be reduced to an
Figure 5.2. An example error run under the PSO memory model.

1: P: L0: x:=1
2: P: L1: y:=1
3: P: L2: $r0:=z$
4: Q: M0: z:=1
5: Q: update(z)
6: Q: M1: $r1:=y$
7: P: update(y)
8: P: update(x)

unguided breadth-first search for optimal fence sets. It will still produce all the optimal fence sets, but will do so in an inefficient manner.

Using the error run \(\tau\) we can implement Analyze-Witness in a more useful way. This can be done by identifying memory access reorderings in \(\tau\), and adding any fence constraint to \(C\) which can prevent those memory access reorderings. Under TSO, this can be done by identifying when stores are delayed past loads by the same thread, and adding fence constraints to \(C\) corresponding to each program location between the store and load. Under PSO, \(W \rightarrow R\) reorderings would be handled in the same way. Additionally under PSO, it would be necessary to identify when different stores from the same thread update to memory in a different order than they appear in control flow, and add both fence and s-fence fence constraints corresponding to each control location between those stores.

For example, consider the PSO error run in Figure 5.2. We can identify that the store \(L0\) is executed on line 1 in the error run, but it effectively takes effect on line 8, where it updates to memory. Similarly, \(L1\) is executed on line 2, but takes effect on line 7. The load \(L2\) is executed on line 3, and takes effect on the same line, since loads occur instantaneously in the abstract machine of PSO. Hence there are three pairs of memory accesses which have been reordered in the thread \(P\), namely \(L0\) with \(L1\), \(L0\) with \(L2\), and \(L1\) with \(L2\). In the other thread, there are no memory access reorderings, as the first instruction is both executed and takes effect before the second instruction.

The reordering of \(L1\) with \(L2\) is a \(W \rightarrow R\) reordering, so a full fence fence is necessary to prevent it. The fence needs to be inserted between \(L1\) and \(L2\) to prevent their reordering, so the witness analysis should add the fence constraint \((\text{fence}, L1)\) to \(C\).

The reordering of \(L0\) with \(L2\) is similar, but here there are two possible choices of location for the fence: immediately after \(L0\) or immediately after \(L1\). So both fence constraints \((\text{fence}, L0)\) and \((\text{fence}, L1)\) are added to \(C\).

The reordering of \(L0\) with \(L1\) is a \(W \rightarrow W\) reordering. Under PSO, we may prevent such a reordering using either a full fence fence or a store fence...
sfence. In both cases, the fence needs to be inserted between \( L_0 \) and \( L_1 \). So we add the fence constraints \( (\text{fence}, L_0) \) and \( (\text{sfence}, L_0) \) to \( C \).

A complete fence constraint set which could be returned by the witness analysis procedure \textbf{Analyze-Witness}, when applied to the PSO error run in Figure 5.2, is thus the following:

\[
C = \{(\text{fence}, L_0), (\text{sfence}, L_0), (\text{fence}, L_1)\}
\]

Notice that since the given run is an error run, any sound fence set must prevent it. We have included in \( C \) all fence constraints that can prevent the error run. Therefore any sound fence set must have a non-empty intersection with \( C \), and so we satisfy the post condition of \textbf{Analyze-Witness}. As a special case, if there is no way to insert fences to avoid all errors in the program, then there are no sound fence sets, and the post condition is trivially satisfied.

In Section 5.4, we will discuss further variations and optimizations that can be made to the witness analysis procedure.

5.3 Proof of Correctness

In this section we will prove the correctness of the fence insertion algorithm in Figure 5.1. This involves proving termination, soundness and completeness.

5.3.1 Termination

\textbf{Theorem 5.2 (Termination).} A call \texttt{Fencins}(\( P, \phi, \kappa \)) will terminate for any \( P, \phi, \kappa \), under the assumption that each call to the parameter functions \texttt{Hits}, \texttt{Reachable} and \texttt{Analyze-Witness} terminates.

\textit{Proof of Theorem 5.2.} We will prove termination by providing a variant for the loop on lines 3-11 in the algorithm in Figure 5.1.

We note first that for any program \( P \) the number of possible candidate fence constraint sets is finite, since there is a finite number of kinds of fences, and a finite number of ways to insert them into \( P \). The variant is then the number of fence constraint sets \( F \) for \( P \) such that \( F \) is a hitting set for \( \text{req} \) and \( F \not\in \text{opt} \).

To see that the variant strictly decreases in each loop iteration, consider any loop iteration, and let \( F \) be the fence constraint set chosen on line 3.

From the way \( F \) was chosen, we know that \( F \not\in \text{opt} \). So if the then-clause on lines 5-6 is executed, and \( F \) is added to \( \text{opt} \), then clearly the variant decreases.

Consider instead the case when the else-clause on lines 7-10 is executed. Then \( \text{opt} \) remains unchanged, but the set \( C \) is added to \( \text{req} \). By the post condition of \textbf{Analyze-Witness}, we know that \( C \) contains no fences which are already in the program \( P \oplus F \). Hence \( C \cap F = \emptyset \). Therefore \( F \) is not a
hitting set of \( \text{req} \) after \( C \) has been inserted, and so we know that the number of hitting sets of \( \text{req} \), as well as the variant, has decreased. This concludes the proof.

5.3.2 Invariant

In order to prove soundness and completeness of the algorithm in Figure 5.1, we need to first prove an invariant for the loop on lines 3-11.

Lemma 5.1 (Invariant). The following two requirements constitute an invariant for the loop on lines 3-11 in the algorithm in Figure 5.1:

1. Every fence set which is sound w.r.t. \( P \) and \( \phi \) is a hitting set of \( \text{req} \), and
2. every fence set in \( \text{opt} \) is optimal w.r.t. \( P, \phi \) and \( \kappa \).

Proof of Lemma 5.1. Both requirements hold trivially before the first loop iteration, since \( \text{req} = \text{opt} = \emptyset \). The rest of the proof consists of showing that the lines 6 and 9, which update \( \text{opt} \) and \( \text{req} \) respectively, maintain the invariant.

Assume that line 6 is executed, and that the invariant holds immediately before. We need to show that \( F \) is optimal w.r.t. \( P, \phi \) and \( \kappa \). First, \( F \) needs to be sound w.r.t. \( P \) and \( \phi \). This holds, because the call \( \text{Reachable}(P \oplus F, \phi) \) must have returned \( \bot \) in order to reach line 6. Next, it must be the case that there is no fence candidate set \( F' \) such that \( F' \) is sound w.r.t. \( P \) and \( \phi \), and \( \kappa(F') < \kappa(F) \). This is known to hold because \( F \) was chosen as one of the cheapest, w.r.t. \( \kappa \), hitting sets of \( \text{req} \), and any sound fence constraint set \( F' \) is a hitting set of \( \text{req} \), by the invariant. Thus executing line 6 does maintain the invariant.

Assume instead that line 9 is executed, and that the invariant holds immediately before. We need to show that any fence constraint set \( F \) which is sound w.r.t. \( P \) and \( \phi \) is a hitting set of \( \text{req} \cup \{C\} \). This follows from the invariant and post condition 1 on \text{Analyze-Witness}. □

5.3.3 Soundness

Theorem 5.3 (Soundness). For any \( P, \phi, \kappa \) and \( F \), if \( F \in \text{Fencins}(P, \phi, \kappa) \), then \( F \) is optimal w.r.t. \( P, \phi \) and \( \kappa \).

Proof of Theorem 5.3. This follows directly from requirement 2 of the invariant proven in Lemma 5.1. □
5.3.4 Completeness

**Theorem 5.4** (Completeness). For any $P$, $\phi$, $\kappa$ and $F$, if $F$ is optimal w.r.t. $P$, $\phi$ and $\kappa$, then $F \in \text{Fencins}(P, \phi, \kappa)$.

**Proof of Theorem 5.4.** Assume for a proof by contradiction that $F$ is optimal w.r.t. $P$, $\phi$ and $\kappa$, and that when the loop on lines 3-11 terminates, $F \notin \text{opt}$. By requirement 1 of the invariant proven in Lemma 5.1, it must be the case that $F$ is a hitting set of $\text{req}$. Hence there exists hitting sets of $\text{req}$, and therefore $\text{Hits}(\text{req}, \kappa) \neq \emptyset$. Next we need to prove that $F \in \text{Hits}(\text{req}, \kappa)$, i.e., that $F$ is a cheapest hitting set of $\text{req}$. Since the loop condition is false, we know that $\text{Hits}(\text{req}, \kappa) \setminus \text{opt} = \emptyset$. Hence $\text{opt}$ is non-empty. Let $c = \kappa(F')$ be the cost of any fence constraint set $F' \in \text{opt}$. Since all elements in $\text{opt}$ are optimal, by requirement 2 of the invariant, they must all have the same cost $c$. Since $\text{Hits}(\text{req}, \kappa) \setminus \text{opt} = \emptyset$, it must be the case that each fence constraint set in $\text{Hits}(\text{req}, \kappa)$ also has the same cost $c$. The set $F$ is optimal by assumption, and therefore also shares the same cost $\kappa(F) = c$. Since $F$ is a hitting set of $\text{req}$ and has the same cost as each hitting set in $\text{Hits}(\text{req}, \kappa)$, we know that $F$ is a cheapest hitting set, and therefore we have $F \in \text{Hits}(\text{req}, \kappa)$. But that gives us the following contradiction: $F \in \text{Hits}(\text{req}, \kappa)$ and $F \notin \text{opt}$ and $\text{Hits}(\text{req}, \kappa) \setminus \text{opt} = \emptyset$, which concludes the proof.

5.4 Optimizations and Variations

5.4.1 Optimizing the Witness Analysis

An imprecise witness analysis produces unnecessarily many alternative fence constraints to prevent a given error run. This affects the performance of the fence insertion procedure in two major ways. First, it causes the number of iterations in the loop on lines 3-11 to increase. In each iteration, a call to the reachability analysis procedure is made. These calls are often very expensive. Second, as the size of the set $\text{req}$ and its constituent fence constraint sets grows, the NP-hard problem of finding cheapest hitting sets starts to become a performance issue. To alleviate both of these performance problems, we should strive to keep the fence constraint sets returned by Analyze-Witness as small as possible. We will discuss two approaches to doing this.

**Run Rewriting.**

A simple way of keeping down the number of fence constraints returned by Analyze-Witness is to perform a rewriting pass on the error run before considering preventive fence placements. Consider again the error run given in Figure 5.2, repeated for convenience in Figure 5.3 (left). We see that the store $L0$ is executed on line 1, but does not update to memory until line 8. However, this delay is not observed by any other event in the run. There is no load
Figure 5.3. The error run $\tau$ (left) in Figure 5.2, rewritten into the error run $\tau'$ (right), to decrease the number of memory access reorderings.

from x, and no other update to x, occurring between line 1 and line 8. This means that the error run $\tau$ in Figure 5.3 (left) can be rewritten into the run $\tau'$ in Figure 5.3 (right), while maintaining the same observable behavior. Hence $\tau'$ is also an error run, which can be analyzed by Analyze-Witness instead of $\tau$. The rewritten run $\tau'$ contains fewer memory access reorderings, and therefore yields the smaller, more efficient, fence constraint set \{$(\text{fence}, \text{L}1)$\} instead of the set \{$(\text{fence}, \text{L}0)$, $(\text{sfence}, \text{L}0)$, $(\text{fence}, \text{L}1)$\} previously computed for $\tau$ in Section 5.2.

Here we used the simple rewriting heuristic that updates are moved back as close as possible to their corresponding stores. For other operational memory models similar rewriting rules can be applied.

Analysis of Shasha-Snir Cycles.

When studying the example in Figure 5.3, we see that we cannot rewrite the run such that the reordering between L1 and L2 disappears, in the same way as we did for L0 and L2. The reason is that the reordering is observed by the update to z on line 5 and the load of y on line 6. Such cases, when memory access reordering is observable, are precisely expressed by cycles in the Shasha-Snir trace corresponding to the error run. This realization leads us to two more possible optimizations of the witness analysis.

First, by computing the Shasha-Snir trace corresponding to the given error run, we can perform the rewriting as described above, but we can identify precisely the reorderings which are observable, by identifying cycles in the Shasha-Snir trace.

Second, we may infer additional fence constraints, which are necessary for soundness, but which do not correspond to reorderings that occur in the given error run. Consider again the rewritten error run $\tau'$ in Figure 5.3. We have previously identified that the fence constraint $(\text{fence}, \text{L}1)$ is necessary to prevent this error run. However, even if this fence is inserted, the same observable erroneous behavior will still be possible in the program, as illustrated by the error run $\tau''$ in Figure 5.4 (left).
An observable behavior corresponding to a Shasha-Snir cycle can often be realized in a run by reordering memory accesses corresponding to any pair of instructions by the same thread occurring in the cycle. Figure 5.4 (right) shows the Shasha-Snir trace corresponding to the error run $\tau'$. We can see that in the run in Figure 5.4 (left), the behavior was realized by reordering the other memory access pair (i.e., $M_0$ and $M_1$) of the Shasha-Snir cycle.

Therefore, when we run the witness analysis on a given error run, we should consider all pairs of instructions by the same thread which appear in a cycle in the Shasha-Snir trace corresponding to the run. So in our example, we would infer the two fence constraints $(\text{fence}, L_1)$ and $(\text{fence}, M_0)$. However, here we should not think of $(\text{fence}, L_1)$ and $(\text{fence}, M_0)$ as being alternatives to each other. If one is necessary, then we know that the other is necessary as well, because they serve to prevent the same behavior. Instead we should treat the analysis of the Shasha-Snir trace conceptually as running the witness analysis multiple times on different error runs: first for the run $\tau'$ and then for the run $\tau''$. Hence we would not return the fence constraint set $C = \{(\text{fence}, L_1), (\text{fence}, M_0)\}$, but instead return two fence constraint sets $C_0 = \{(\text{fence}, L_1)\}$ and $C_1 = \{(\text{fence}, M_0)\}$. Both the set $C_0$ and $C_1$ would then be added separately to $\text{req}$ on line 9 in the fence insertion algorithm. This has the effect that the fence insertion algorithm constrains the search for optimal fence sets even more, and it allows even faster convergence at the optimal fence sets. With this technique it is sometimes even possible to converge on the optimal fence sets after performing fewer iterations in the fence insertion algorithm than the number of fences in the optimal set.

---

This would of course require us to change the signature of $\text{Analyze-Witness}$ to return a set of sets of fence constraints. Line 9 in Figure 5.1 would change to $\text{req} := \text{req} \cup C$. 

---

Figure 5.4. Left: Another error trace $\tau''$ with the same observable behavior as $\tau$ and $\tau'$ in Figure 5.3, despite a fence being added between $L_1$ and $L_2$. Right: The Shasha-Snir trace corresponding to the error runs $\tau$ and $\tau'$. The error run $\tau''$ shares the same Shasha-Snir trace, but with the addition of an extra fence.
Furthermore, analyzing Shasha-Snir traces instead of runs, also allows us to employ the fence insertion algorithm for memory models which are axiomatic rather than operational, such as the model of POWER in Section 3.3.

5.4.2 Approximate Reachability Engines

In the discussion thus far we have implicitly assumed that the reachability engine \textbf{Reachable} performs a precise analysis: terminating, sound and complete. However, it is possible to also parameterize the fence insertion algorithm with a reachability engine which is approximate. The notion of a \textit{sound} fence constraint set should then be reinterpreted as meaning a fence constraint set such that the reachability engine does not report an error for the fenced program.

With an \textit{under-approximate} reachability engine, this has the effect that the fence insertion algorithm returns fence sets that are strong enough to prevent all errors that the reachability engine could identify. The returned fence constraint sets will not contain any superfluous fences, but may not be strong enough to actually guarantee correctness. This is the approach taken by Liu et al. in [37], where this algorithm was first published.

With an \textit{over-approximate} reachability engine, the effect is that the fence insertion algorithm returns fence constraint sets that are strong enough to guarantee correctness of the program, but may contain superfluous fences which have a detrimental effect on program performance. Additionally, if the over-approximate reachability engine is too imprecise, it may discover a spurious error under the SC memory model, and then incorrectly report that the program cannot be corrected with fences.
In this text, we have introduced the background knowledge relevant to understanding the attached articles. We have discussed concurrent programs in general, and their semantics under sequential consistency. We have examined the behaviors caused by relaxed memory, and seen examples of the kinds of architectural implementation choices that motivate these models. We have examined in some detail the formal semantics of concurrent programs under TSO, PSO, and POWER. For analyzing concurrent programs, we have looked particularly into systematic testing with stateless model checking and dynamic partial order reduction. We have seen how these techniques can be applied under SC, and how the techniques can be adapted to relaxed memory. We have also looked into how analysis techniques can be used to support automatic inference of memory fences, which are necessary to guarantee correctness under relaxed memory.

In the attached articles we introduce a number of new techniques for analyzing concurrent programs under relaxed memory. In Paper I, we introduce a sound and complete reachability analysis for finite state programs running under the TSO memory model. The technique is combined with automatic fence inference, and available in the open source tool Memorax. In Paper II, we introduce a technique where predicate abstraction is used together with buffer abstraction to produce an over-approximative reachability analysis for integer programs running under TSO. The technique can then use an optional CEGAR loop to regain preciseness, but lose the guarantee for termination. This technique is also paired with fence inference, and implemented in Memorax. In Paper III, we introduce the notion of chronological traces, and show how they can be used to apply stateless model checking to concurrent programs running under TSO and PSO. This technique is an under-approximation, since it bounds the length of the explored program runs. But this allows us to apply the technique to more realistic programs, written in C without the need for a human analyst to model the program in a special purpose modelling language. This could mean that the technique can be more readily applied directly to unit tests.
written for actual applications. In Paper IV, we apply stateless model checking to the far more relaxed POWER memory model. In order to do so we introduce a new operational model for POWER, based directly on the axiomatic model [14]. We then propose a new stateless model checking algorithm which is spiritually close to dynamic partial order reduction, but adapted to our operational model where semantics are governed by events’ parameter choices rather than only by the scheduling of memory accesses. The techniques of both Papers III and IV are implemented, and available as open source, in the tool Nidhugg.

Future Work.
The Papers III and IV are a step toward more usable techniques compared to Papers I and II. In order to further make the techniques useful to someone who is writing application or system code the techniques should be adapted to work for the memory models of higher level languages such as C++ or Java. They should then be adapted to detect both specification errors and cases where the code has undefined behavior (such as data races in C++ and Java). Furthermore, it would be appropriate to integrate the techniques into existing unit testing frameworks such that they are easier to employ in existing projects and existing work flows.

Another direction which would improve usefulness is to add support for data-nondeterminism using ideas from symbolic execution and concolic testing.

In order to improve scalability, it would be useful to consider coarser equivalence classes for runs than those given by Shasha-Snir traces. These coarser classes could e.g. aim to be independent of coherence order.
Summary in Swedish: Verifiering av program under svaga minnesmodeller


Behovet att kombinera alla dessa designkrav gör att utveckling av effektiva parallella program är svårt, och att fel (buggar) lätt kan introduceras i programmen. Parallellism-relaterade fel är ofta särskilt svåra att upptäcka och rätta till. Det är för att i ett program som innehåller parallellism-relaterade fel, så kan det
vara sällan och oförutsägbart som felen faktiskt manifesterar sig. D.v.s. även om ett program innebär ett fel som kan visa sig i ett specifikt testfall så finns ingen garanti för att man faktiskt kommer att se felet, även om man kör precis det testfallet många tusen gånger. Även om ett fel uppstår väldigt sällan kan det vara allvarligt när det väl visar sig. Det gäller särskilt för säkerhetskritiska system, men även i kommersiella sammanhang där fel kan vara dyra.


Att konstruera korrekt program som fungerar under svaga minnesmodeller kräver att programmeraren förstår minnesmodellen, och noggrant specifiserar vilka minnesåtkomster som måste hållas ordnade i förhållande till varandra. Eftersom svaga minnesmodeller gör programmering svårare så ökar risken för fel ytterligare.

För att upptäcka och rätta till fel, och för att få en högre säkerhet vad beträffar programs korrekthet kan automatiska tekniker för verifiering och systematisk testning vara användbara. Automatiska tekniker för verifiering strävar efter att automatiskt producera matematiska bevis för korrekthet hos program. Systematisk testning är rigorösa, automatiska testtekniker som används för att köra test som ger garantier om att täcka en viss del av alla fall.

där tillstånd kategoriseras som otillåtna om konsumenten läst data från minnet utan att producenten skrivit den datan.

Den här avhandlingen handlar om automatisk verifiering och systematisk testning av parallella program. I synnerhet behandlar arbetet hur man kan utveckla sådana tekniker som tar hänsyn till fel som orsakas av de svaga minnesmodellernas omordningseffekter. De bifogade artiklarna presenterar fyra tekniker. I artikel I presenterar vi en sund och fullständig analys-teknik för program under minnesmodellen TSO med ändlig tillståndsrymd. I artikel II presenterar vi en teknik som klarar av heltalsprogram under TSO med oändlig tillståndsrymd, men där analysen inte garanterat terminerar. I artiklarna III och IV presenterar vi systematiska testtekniker för program under minnesmodellerna TSO, PSO och POWER.
[38] Discussion on the Linux Kernel Mailing List about an optimization to a mutual exclusion unlock primitive under x86. https://lkml.org/lkml/1999/11/20/76.
My Contributions

**Paper I.** For this paper I implemented the technique and ran the experiments. I participated in discussions developing the theory. I later rewrote the implementation and made it publicly available as the tool Memorax [33].

**Paper II.** For this paper I participated in discussions developing the theory, implemented the technique and ran the experiments. I have integrated this technique in Memorax [33] as well.

**Paper III.** For this paper I developed the theory in discussion with co-authors, implemented the technique, ran part of the experiments, and wrote most of the article as well as the proofs in the extended technical report [3]. The implementation is publicly available at [34].

**Paper IV.** For this paper I developed the theory in discussion with co-authors, implemented the technique, ran the experiments, and wrote most of the article and the proofs. The implementation is publicly available at [34].
A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)