AUTOMATING C++ EXECUTION EXPLORATION TO SOLVE THE OUT-OF-THIN-AIR PROBLEM

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

Simon Cooksey

May 2021

Abstract

Modern computers are marvels of engineering. Customisable reasoning engines which can be programmed to complete complex mathematical tasks at incredible speed. Decades of engineering has taken computers from room sized machines to near invisible devices in all aspects of life. With this engineering has come more complex and ornate design, a substantial leap forward being multiprocessing. Modern processors can execute threads of program logic in parallel, coordinating shared resources like memory and device access. Parallel computation leads to significant scaling of compute power, but yields a substantial complexity cost for both processors designers and programmers.

Parallel access to shared memory requires coordination on which thread can use a particular fragment of memory at a given time. Simple mechanisms like locks and mutexes ensure only one process at a time can access memory gives an easy to use programming model, but they eschew the benefits of parallel computation. Instead, processors today have complex mechanisms to permit concurrent shared memory access. These mechanisms prevent simple programmer reasoning and require complex formal descriptions to define: *memory models*.

Early memory model research focused on weak memory behaviours which are observable because of hardware design; over time it has become obvious that not only hardware but compilers are capable of making new weak behaviours observable. Substantial and rapid success has been achieved formalising the behaviour of these machines: researchers refined new specifications for shared-memory concurrency and used mechanisation to automate validation of their models. As the models were refined and new behaviours of the hardware were discovered, researchers also began working with processor vendors – helping to inform design choices in new processor designs to keep the weak behaviours within some sensible bounds. Unfortunately when reasoning about shared memory accesses of highly optimised programming languages like C and C++, deep questions are still left open about how best to describe the behaviour of shared memory accesses in the presence of dependency removing compiler optimisations. Until very recently it has not been possible to properly specify the behaviours of these programs without forbidding

optimisations which are used and observable, or allowing program behaviours which are nonsense and never observable.

In this thesis I explore the development of memory models through the lens of tooling: taking at first an industrial approach, and then exploring memory models for highly optimised programming languages. I show that taming the complexity of these models with automated tools aids bug finding even where formal evaluation has not. Further, building tools creates a focus on the computational complexity of the memory model which in turn can steer development of the model towards simpler designs.

We will look at 3 case studies: the first is an industrial hardware model of NVIDIA GPUs which we extend to encompass more hardware features than before. This extension was validated using an automated testing process generating tests of finite size, and then verified against the original memory model in Coq. The second case study is an exploration of the first memory model for an optimised programming language which takes proper account of dependencies. We build a tool to automate execution of this model over a series of tests, and in the process discovered subtleties in the definitions which were unexpected – leading to refinement of the model. In the final case study, we develop a memory model that gives a direct definition for compiler preserved dependencies. This model is the first model that can be integrated with relative ease into the C/C++ programming language standard. We built this model alongside its own tooling, yielding a fast tool for giving determinations on a large number of litmus tests – a novelty for this sort of memory model. This model fits well with the existing C/C++ specifications, and we are working with the International Standards Organisation to understand how best to fit this model in the standard.

Acknowledgements

My heartfelt thanks go to my supervisor, Professor Mark Batty, for his patience and excellent guidance during my PhD. I also thank my mother for her consistent belief in me, and my work. Thanks Mum. Finally, I write this thesis in fond memory of Jeremy Harle.

Contents

Chapter 1

Introduction

Modern computers are the result of unimaginable engineering effort over many decades. They are the among the most refined artefacts of human endeavour: the coordinated design of thousands has resulted in a construction more complex than any object built before. It is no surprise then, that aspects of how computers work are unknown even to their designers — leading to several fields of research which uncover remarkable behaviours and unintelligible bugs.

As CPU vendors have found it more and more challenging to continue an inexorable increase in clock speed, they have instead looked towards hardware parallelism to drive additional performance. This has allowed consumer CPUs to scale from machines which can execute a single instruction at a time to a complex architecture permitting dozens of concurrent threads of execution which may all access the same pool of shared memory. These multiprocessor CPUs are now commonplace in every new computer sold, and the vast majority of smartphones.

In an attempt to tame the complexity of reasoning about computer behaviour, practitioners have been developing formal abstract descriptions of processors. One such aspect which has been formally abstracted is the *memory model*: a description of what values can be returned by loads and stores. In this thesis, we will focus on so-called weak memory behaviours which arise from the complex interactions of multiprocessors sharing memory. This area of research has had significant attention over the years, particularly in the last decade – and significant strides have been made towards usable and comprehensible memory models for parallel processors.

Unfortunately, the programming languages at the core of all modern computer programs suffer even more so with weak memory behaviours than the hardware that they sit upon. Aggressive optimisations interact in unexpected ways with weakly ordered hardware, and existing approaches to formalise programming language memory models have yielded unsatisfying results.

In particular, the ISO C and C++ committee provides the authoritative definition of the C and C++ languages respectively, but their definition of memory semantics is currently incomplete (Batty et al. 2015).

1.1 Weak Memory Consistency

Modern computers forgo simplicity in favour of performance. Programming languages and modern computer architectures are built with aggressive optimisations in mind, taking many forms. Compilers re-write programs making semantic preserving changes which reduce execution time – such as re-using previously computed values, or recognising that some control flow checks are unnecessary. Processor designers similarly optimise their architecture: processors might over-fetch data, or collect writes to send to system memory in blocks, amortising fixed write overheads across several operations. Unfortunately the correctness of these optimisations relies on imposing restrictions on programmers, and the assumptions that they might make about how a piece of code behaves can be wildly inaccurate. What is worse, is that many of the behaviours that result from these optimisations can only be observed occasionally if two threads sharing memory line their accesses up in time, a rare occurrence. A memory consistency model (or memory model) formally describes these behaviours, giving programmers a description of how loads and stores will behave. Memory models can be at the hardware level (i.e. x86, ARM, POWER, NVIDIA PTX) (Podkopaev, Lahav and Vafeiadis 2019; Sewell et al. 2010; Pulte et al. 2018; Sarkar et al. 2011; Lustig et al. 2017), describing how machine load and store operations work; or at the programming language level (i.e. C++, Java) (Batty et al. 2011; Manson, Pugh and Adve 2005), describing how assignments to variables behave. A memory model can be used to examine the possible behaviours of some code without having to exhaustively execute it, negating the worry that direct testing of code will not hit the sometimes one in one-billion odds of observing some architectural weakness (Alglave et al. 2011; Sewell et al. 2010).

1.1.1 Litmus Tests

Added explanation of litmus tests.

An essential mechanism for understanding memory models are so-called *litmus tests*. Litmus tests are short programs designed to probe individual facets of the memory system to reveal strange behaviours. Each test comes with an expected outcome, representing the interesting behaviour. Litmus tests are normally top-level parallel with some initialisation prefixing all threads. Litmus tests are typically written in a simple grammar exposing loads, stores, conditional

control flow, and parallel composition. Additional features may be present as the memory model requires, for example, some memory models (Kang et al. 2017) support system calls, atomic read-modify-write operations (Lahav et al. 2017), and memory dereferencing (Jeffrey and Riely 2016). Throughout this thesis we will refer to litmus tests in a language extended from P below, where `r` ranges over registers, M ranges over simple arithmetic expressions (including registers, but not memory locations), and `x` ranges over memory locations.

$$T ::= \texttt{r := x} \mid \texttt{x :=} M \mid \textbf{if}(\texttt{r} = M)\{T_1\}\{T_2\} \mid T_1\,;\, T_2$$
$$P ::= \texttt{x := 0;}\ \ldots\ ;\ \texttt{z := 0;}\ (T_1 \parallel T_2 \parallel \cdots \parallel T_n)$$

This language explicitly separates memory loads (`r := x`) from memory stores (`x :=` M).

For example, Figure 1 displays the Store Buffering (SB) litmus test. SB comprises two threads which store a value to a global address, and then read a value from a different address. It demonstrates a weak behaviour admitted by an architectural feature of x86 where the weak behaviour $\texttt{r}_1 = \texttt{r}_2 = 0$ can be observed. A naïve programmer might not expect this execution, because no interleaving of the operations of threads 1 and 2 would yield a state where both the reads can read from the initialisation of `x` and `y`. In practice this execution is allowed because of behaviours introduced by optimisations in the x86 architecture, the exact mechanism for which is explained in §2.1.2. In fact a compiler optimisation can also introduce this behaviour: a compiler can observe that the write to `x` is unrelated to the subsequent read from `y`, and re-order the accesses.

Initially: `x := 0; y := 0;`

`x := 1;`	‖	`y := 1;`
$\texttt{r}_1$ `:= y`	‖	$\texttt{r}_2$ `:= x`

Figure 1: Store buffering litmus test. The weak behaviour would be observing that $\texttt{r}_1 = \texttt{r}_2 = 0$.

The common patterns of litmus tests are those which show certain kinds of multi-thread synchronisation. A popular set of these tests comes from the supplemental material with the POWER memory model (Maranget, Sarkar and Sewell 2012). It lists the most common synchronisation patterns and where a weak behaviour is permitted, and annotates each example with what additional POWER instructions must be added to forbid the weak behaviour.

Initially: x := 0; y := 0;	
r_1 := y;	r_2 := x;
x := 1	y := 1

Figure 2: Load buffering litmus test (LB).

Initially: x := 0; y := 0;	
r_1 := y;	r_2 := x;
if($r_1 = 1$){x := 1}	**if**($r_2 = 1$){y := 1}

Figure 3: Load buffering litmus test, with data dependencies (LB+deps).

1.2 The Out-Of-Thin-Air Problem

The current formal definitions of Java (Manson, Pugh and Adve 2005) and C/C++ (ISO/IEC 2010) allow program executions which appear to summon values *Out of Thin Air* (OOTA) (Pugh 1999; Batty et al. 2015). To explore OOTA executions, we will consider a handful of programs and how they might execute. Figure 2 presents the Load Buffering (LB) litmus test which is a canonical reference for exploring OOTA (Batty et al. 2015; Kang et al. 2017; Pichon-Pharabod and Sewell 2016; Jeffrey and Riely 2016; Paviotti et al. 2020). LB is a two thread litmus test in which each thread reads from a shared location and then writes to a different shared location.

In this case, we might wonder if we can observe the weak behaviour where $r_1 = r_2 = 1$. Either because of architectural features that allow re-ordering of the apparently unrelated loads and stores of each thread, or compiler optimisations which do the same. Indeed, as some hardware does this re-ordering (Maranget, Sarkar and Sewell 2012; Deacon and Alglave 2019) it is necessary to forbid the weak behaviour. Fences incur a performance penalty, so doing so is undesirable, and indeed C/C++ compilers do not emit them: allowing the weak behaviour. All other outcomes of LB could be explained by executing the program one instruction at a time without re-ordering.

Let's consider a modification of this program, which should prevent the weak execution. In Figure 3 we have introduced a condition on each thread, contingent on the value read in the operation before the store. In this case a compiler could not re-order the operations, as they now *are* related.

So the memory model must allow the weak execution in the first program, LB, but forbid the weak execution of the second program, LB+deps. Informally this execution must be forbidden because calculation of non-zero values in `r1` and `r2` is self-referential: one must first read the

Initially: x := 0; y := 0;

r_1 := y;	r_2 := x;
if($r_1 = 1$) {	**if**($r_2 = 1$) {
x := 1	y := 1
} **else** {	} **else** {
x := 1	y := 1
}	}

Figure 4: Load buffering litmus test, with false data dependencies (LB+false-dep).

Initially: x := 0; y := 0;

r_1 := y;	r_2 := x;
x := r_1	y := r_2

Figure 5: Load buffering litmus test, with data dependencies (LB+datas). The OOTA behaviour would be observing that $r_1 = r_2 = 42$.

value, to write it, and to read it the other thread must have written it – so must have read the value... and so on. Naïvely we might try and capture this kind of cycle in value calculation. To explore this idea a little further, let us consider another program, LB+false-deps, printed in Figure 4.

In LB+false-deps a compiler can trivially optimise the program back to the original LB (Figure 2) as regardless of the control flow choice, the assignment of x := 1 will happen regardless. As a result, the memory model must allow the outcome $r_1 = r_2 = 1$. Thus simply observing in a single execution that there's some cyclic calculation of the value for r_1 and r_2 does not imply the execution should be forbidden. Things get worse, though. Consider the program in Figure 5: it would be peculiar for $r_1 = r_2 = 42$: the value 42 does not appear in the code, and cannot be calculated from the values in the program. Yet, the C/C++ memory model allows the execution for the same reason it allowed $r_1 = r_2 = 1$ for LB+false-deps.

Unfortunately this is because the C/C++ memory model cannot detect which dependencies are merely *syntactic*, i.e. dependencies which can be eliminated by compiler optimisation, and which dependencies are are *semantic* (McKenney et al. 2016), i.e. dependencies which materially affect the execution of the program. This inability to distinguish LB and LB+datas is known as the *Out-Of-Thin-Air Problem*, and is the focus of much of the content of this thesis. OOTA is a big problem: consider that the data stored in x of LB+deps is actually a pointer which should only be handed over when a certain security constraint is met, perhaps it points to some secret

data. Having a memory model which allows thin-air executions means that we cannot prove that our LB program does not gain access to the secret memory – even though in reality it could not execute like that with any combination of hardware and compiler. This ability to summon illegal pointers is a barrier to proving safety properties like no use-after-free, or some kinds of buffer over runs.

1.3 Contributions

This work is a step towards a new standardised version of the C/C++ memory model which permits aggressive compiler optimisations and forbids thin-air executions. This thesis makes the argument that tools are essential to building and understanding complex memory model. Ultimately a tools based approach has allowed us to define a new memory model for C/C++ which shows a lot of promise (Giroux 2019).

Understanding concurrency in C/C++ is something that has driven significant research effort over several years (Kang et al. 2017; Pichon-Pharabod and Sewell 2016; Jeffrey and Riely 2016; Chakraborty and Vafeiadis 2019), and part of what makes it so complex is that a *good* definition for the C/C++ memory model must be sensitive to two things. First, it must be a mathematically sound and usable way to prove useful properties about concurrent programs. Secondly, it must fit with the existing C/C++ standard without widespread rewrites to substantial portions of the standard text. The C/C++ standard is a contract between language implementers and programmers, so re-writing large sections of that is not a noble thing nor remotely likely to be approved. While reasoning about thin-air free memory models, we find that tools to execute these models are the ultimate mechanism to detect bugs and explain the model to others.

Progress towards this goal took place in stages, first looking at axiomatic models for hardware execution, then examining thin-air models, before finally proposing a solution of our own. Concretely, the thesis contributions are as follows:

- A better understanding of memory system features in NVIDIA PTX
- The first tool to automatically evaluate a thin-air free memory model over a suite of litmus tests
- A denotational model of weak memory consistency which provides a definition of semantic dependency
- Integration of semantic dependency into a full-fat C/C++ memory model to provide a thin-air free C/C++ memory model

1.4 Thesis Structure

In this thesis we will see why tools are essential for tackling the complexity of building and reasoning about weak memory consistency models. This thesis contains 3 case studies, each using tools to aid building concurrency models. These case studies work from the ground up, starting at hardware and working towards a sensible model of C/C++ which permits optimisations, but does not allow thin-air free executions.

NVIDA PTX In the first case study, Chapter 3, we look at the NVIDIA PTX memory model which is the hardware model for NVIDIA graphics cards. We extend PTX to cover more hardware features, and introduce new relaxations. Tools are used to validate that these changes do not interfere with the previous definition, and to generate exhaustively programs which might execute in a weakly consistent manor on the hardware.

An Event Structures Model of Relaxed Memory. In the second case study, Chapter 4, we look at the thin-air free model for Java presented by Jeffrey and Riely (2016). They present a formula for exploring the possible states of a program execution, yielding a novel thin-air free semantics for Java. In this instance we develop a tool to automatically evaluate the model over a suite of litmus tests. This process uncovered a bug with their model, and led to a fix – even despite existing mechanised proofs.

Modular Relaxed Dependencies. In the final case study we develop a thin-air free C++ semantics which covers much of the C++ language and forms a candidate to drop into the existing C++ specification. Modular Relaxed Dependencies (MRD), defined in Chapter 5 solve the thin-air problem by decoupling the calculation of dependencies which are semantic from the syntax of the program. We then use this semantic dependency calculation to build a fully featured C/C++ model, Chapter 6, the existing axiomatic model provided by (Lahav et al. 2017) and re-expressed by (Podkopaev, Lahav and Vafeiadis 2019). The construction of MRD has a blended approach of execution exploration and per-execution axiomatic restriction, the development of which was strongly informed by the concurrent building of a tool for automatic evaluation over many litmus tests. Thus, in Chapter 7, we analyse the semantics in the context of other thin-air free models, using automatic evaluation to mechanically derive permitted executions for a suite of litmus tests in the literature. This was presented as a paper at ESOP 2020 and has formed a working paper for the ISO C++ Standards Committee.

Chapter 2

Memory Models: State-of-the-art

Memory model development has seen substantial academic interest over the past decade. A focus on formal specification and empirical model validation have resulted in large strides forward for reasoning about software behaviour. In this chapter we will introduce memory model concepts and tour the key academic developments in memory model formalisation over the last few years. We will start with a model for a simple concurrent machine, and subsequently introduce more modern and weakly ordered hardware. Once we have considered hardware we will move on to look at programming language models, the Out-Of-Thin-Air (OOTA) problem, and some existing solutions to OOTA.

Programmers need to understand how their code will execute on real hardware in order to write bug-free software. To talk about memory models concretely, we will first pin down a couple of definitions. *Memory models* are precise specifications of what values can be returned by loads. *Consistent* executions of a program are those which are allowed by the memory model, i.e. all the loads returned in the execution satisfy the constraints of the memory model. These consistent executions provide bounds on program behaviour, but a given allowed execution may not be observable in practice. A memory model's *strength* describes how many executions are permitted for a given program, and comparing memory models can be done with quantitative comparisons of their strength. The stronger a memory model, the fewer executions it allows, the weaker a model, the more executions it allows. Programming language designers and processor architects design their systems with particular programming idioms and optimisations in mind. The strongest possible memory model would permit no executions of any programs, while the weakest would allow any execution of any program. These extremes of strength and weakness are not useful, so memory models of interest sit in the spectrum in-between. Stronger memory

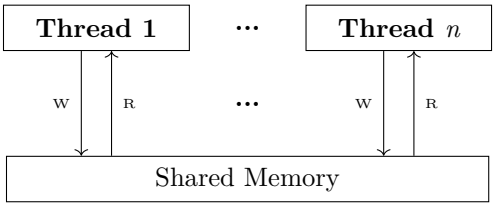


Figure 6: SC block diagram

models typically allow more use of more straightforward programming idioms, whereas weaker models allow more aggressive optimisation of code. In this literature review we will explore important memory models for concurrency, from the strongest model of Sequential Consistency (§2.1.1) to the weakest models of ARM (§2.1.3) and C++ (§2.1.4).

2.1 Industrial models

2.1.1 Sequential Consistency

A simple model of memory behaviour in a concurrent setting is the model of *Sequential Consistency* (SC). This model, described by Lamport (1979), presents a simple "interleaving semantics" of memory accesses, where each thread in the system takes one step at a time modifying a single global shared memory. Lamport concisely describes a machine as sequentially consistent if:

> *... the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
>
> – Lamport (1979)

A simple block diagram of an SC machine is shown in Figure 6. The SC machine has n threads, each with arrows to indicate their data paths for reads (R) and writes (W) to a single shared memory. Each thread can read or write the shared memory one at a time, which creates the interleaving. As such, there is a single global order of stores to memory which is the same to all observers. Let's consider SB again (Figure 7), and all of the possible SC executions of that program: i.e. all of the sequential orders that the operations could be performed in Lamport (1979).

Initially: $\mathtt{x} = \mathtt{y} = 0$

Thread 1		**Thread 2**
`x := 1;`	‖	`y := 1;`
$\texttt{r}_1$ `:= y`	‖	$\texttt{r}_2$ `:= x`

The interleaved programs can be displayed conveniently as a single stream of operations which are executed left to right. There are six valid interleavings: two simple cases where Thread 1 fully executes, then Thread 2 and vice versa; and 4 other executions where operations alternate between the two threads. The first case where Thread 1 fully executes then Thread 2 would be represented as `x := 1; r1 := y; y := 1; r2 := x` – where 1 is written to `x`, 0 is read from `y`, 1 is written to `y` and finally 1 is read from `x`. This yields an execution where we observe $\texttt{r}_1 = 0 \wedge \texttt{r}_2 = 1$. Each execution and the final state of the registers is enumerated in the table below.

	Final State	
Sequentialised code	$\texttt{r}_1$	$\texttt{r}_2$
`x := 1; r1 := y; y := 1; r2 := x`	0	1
`y := 1; r2 := x; x := 1; r1 := y`	1	0
`x := 1; y := 1; r1 := y; r2 := x`	1	1
`y := 1; x := 1; r1 := y; r2 := x`	1	1
`x := 1; y := 1; r2 := y; r1 := x`	1	1
`y := 1; x := 1; r2 := y; r1 := x`	1	1

Under sequential consistency the final state $\texttt{r}_1 = 0 \wedge \texttt{r}_2 = 0$ is not observable, which matches the naive assumption that loads and stores can't be reordered. Unfortunately for programmers SC behaviour is not something one can take for granted, and real computer systems are more complex. As far back as 1972 with the IBM System 370/158MP (Maranget, Sarkar and Sewell 2012), processors have exhibited non-sequentially consistent behaviour: where program executions do not correspond to an interleaving of each thread's instructions. Further, key architectures available today (ARM, x86, POWER, RISC-V…) are also more weakly consistent. To understand this better, let us first examine x86.

2.1.2 x86-TSO

There is an inherent tension between hardware architects and software engineers: hardware architects want to specify the weakest possible architecture, to permit more aggressive optimisation

in the future, while software engineers need to write well specified programs which are built on guarantees provided by the underlying hardware.

When trying to understand the precise behaviour of memory accesses on x86, Sewell et al. (2010) found contemporary specifications from hardware vendors to "*leave key questions ambiguous*". To better understand how weak memory in x86 can affect concurrent programs it is possible to construct small programs whose behaviours are non-SC. Where study of the specifications reveals ambiguity, tests can be carefully crafted to help answer the questions raised. Observation of a weak behaviour on hardware is enough to evidence a weak behaviour is possible, and not being able to observe a behaviour allows researchers to ask questions of hardware architects about specific assumptions of the architecture's design. It is not a guarantee that not observing a weak behaviour means that the architecture will always forbid it, either. Architects may implement optimisations which result in new observable weak memory behaviours without changing the specification of their machines.

Weak behaviours and Litmus Tests

To explore x86 we begin as Sewell et al. (2010) did and investigate some short programs – known as *litmus tests* – which shed light on the weak behaviours.

Store Buffering. Store Buffering (SB) is shown in Figure 7, and is a key example that demonstrates a non-SC behaviour on the x86 architecture – it is the same example we examined under Sequential Consistency (§2.1.1), but instead presented in concrete x86 assembly. Figure 8 gives a high-level block diagram for memory accesses in x86: arrows indicate dataflow between hardware threads and memory. The write buffer is a cache for stores to memory and is the implementation of an architectural optimisation which reduces the latency of local accesses to a memory location and reduces bandwidth requirements to shared memory. This optimisation introduces a new weak behaviour: the execution of interest in Figure 7 is one where `EAX` $==$ $0 \wedge$ `EBX` $==$ 0, and does not correspond to an SC execution of the program – there is no interleaving of operations which would leave the register file in this state.

Each hardware thread's write buffer is a read/write cache which can "short-circuit" accesses to shared memory when there has been a recent store to a given location in the same thread. Considering each thread in turn: Thread 1 first stores value 1 into `x`, which is cached in a write buffer; Thread 2 similarly stores 1 in `y` – and is also cached in the local write buffer. Then each thread performs a load, Thread 1 does not have a value for `y` in its local write buffer, so reads the now-stale value 0 from shared memory. Similarly, Thread 2 does not have a value for `x` in its

Initially: x = y = 0

Thread 1	Thread 2
MOV [x], 1;	MOV [y], 1;
MOV EAX, [y];	MOV EBX, [x];

Figure 7: Store buffering in x86 assembly. Observing EAX == 0 ∧ EBX == 0 after this program executes corresponds to a weak memory behaviour.[1]

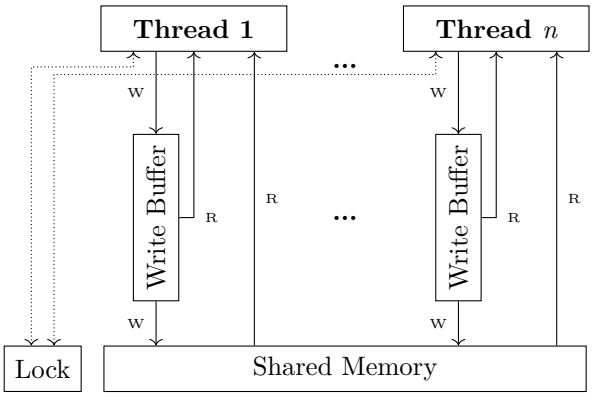


Figure 8: x86-TSO block diagram

write buffer and also reads a stale value of 0 from shared memory.

Message Passing. Message Passing (MP), shown in Figure 9, reflects a common notify-wait programming pattern. A producer thread does some computation to calculate a value to store into x, and then notifies a waiting thread that the value is ready by setting a flag in y. The write buffer propagates values to shared memory in order, meaning that the write of the flag cannot "overtake" the write of the data. As such, EAX == 1 ∧ EBX == 0 is forbidden. This test shows that local store-store ordering is respected globally. As we will see later in §2.1.3 other architectures are weaker and require additional operations to preserve ordering between stores. This makes MP a useful test to contrast the behaviours present in x86 with, for example, ARM.

[1] Registers are local to each thread, but it is clearer to use unique register names to disambiguate final states.

Initially: x = y = 0

Thread 1	Thread 2
MOV [x], 1;	MOV EAX, [y];
MOV [y], 1;	MOV EBX, [x];

Figure 9: Message passing in x86 assembly. Observing EAX == 1 ∧ EBX == 0 after this program must be forbidden.

Initially: x = y = 0

Thread 1	Thread 2
`MOV [x], 1;`	`MOV [y], 1;`
`MFENCE;`	`MFENCE;`
`MOV EAX, [y];`	`MOV EBX, [x];`

Figure 10: Store buffering in x86 assembly with `MFENCE` instructions. Observing `EAX` $== 0 \wedge$ `EBX` $== 0$ is now forbidden.

Restoring Order

It is of course undesirable to always have weak memory effects, as they can break fundamental building blocks of shared memory communication, such as Store Buffering (Figure 7). As such, in x86 there are operations available which prevent the store buffers from apparently re-ordering stores. x86 provides an instruction which flushes the store buffers to system memory, providing the following guarantee:

> *[`MFENCE` p]erforms a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the `MFENCE` instruction. This serializing operation guarantees that every load and store instruction that precedes the `MFENCE` instruction in program order becomes globally visible before any load or store instruction that follows the `MFENCE` instruction.*
>
> — Intel 64 and IA-32 Architectures Software Developer's Manual (Intel Corporation 2016)

Thus in x86 inserting `MFENCE` instructions on each thread of the original store buffering program will forbid the weak execution. We will call this new program SB+fences, and it is printed in Figure 10. By serialising the loads-from-memory and the stores-to-memory we can ensure there is a simple interleaving, and we restore sequentially consistent behaviour. That is, with `MFENCE` we will forbid the outcome `EAX` $== 0 \wedge$ `EBX` $== 0$.

Validation. A formal model of x86 is useful only if it can be trusted as a faithful model of software running on the platform, and without vendor endorsement the only way to build trust is exhaustive execution of tests. Alglave et al. (2011) provide a tool, `litmus`, for running litmus tests against hardware. `litmus` takes care to use fine control of thread start and processor state to give hardware the best chance to exhibit rare weak memory behaviours.

Using `litmus` does build substantial confidence that the model is accurate – discrepancies would be obvious, but `litmus` does not show a model to be *complete*: it cannot provide a guarantee that all weak behaviours are covered. Nor can `litmus` show a model to be *sound*: it

Initially: $\texttt{x} = \texttt{y} = 0$

Thread 1		**Thread 2**
$\texttt{x} := 1;$	$\parallel$	$\texttt{r}_1 := \texttt{y};$
$\texttt{y} := 1$	$\parallel$	$\texttt{r}_2 := \texttt{x}$

Figure 11: Pseudocode Message Passing litmus test. Observing $\texttt{r}_1 = 1 \wedge \texttt{r}_2 = 0$ is permitted on ARM.

cannot provide a guarantee that all weak behaviours permitted by the model can be observed on the hardware. `litmus` is an empirical tool, and as Sewell et al. (2010) note: "there may be tests where x86-TSO allows some final state that cannot be observed in practice". Despite this, `litmus` provides a useful lens into processor behaviour – allowing the execution of manually selected interesting tests, and as we will see later in section §2.2.1, the execution of tests automatically generated from formal specifications of memory models.

2.1.3 ARM

ARM designs a family of architectural specifications which are implemented by vendors like Broadcom, Qualcomm, and Apple. Some of these designs are small-scale single-threaded embedded platforms, but others are large application processors which run in consumer electronics, mobile phones, and servers. These larger processors support multiprocessing, and exhibit a weaker memory model than x86. The ARM memory model is provided in ARM's architecture manual as an english language specification (ARM 2020, 2014), and a formal mathematical definition in the form of a CAT model (Deacon and Alglave 2019; Alglave, Maranget and Tautschnig 2014). CAT models will be explored in greater detail in §2.2.1.

To see how ARM is weaker than x86, we can look at MP again. This time MP is presented in Figure 11. Unlike the previous presentation of MP, the litmus test now is rendered in psuedocode. ARM is a RISC architecture which requires more instructions to express programs, so the psuedocode treatment keeps presentation clear.

In MP on x86, executions where $\texttt{r}_1 = 1 \wedge \texttt{r}_2 = 0$ are forbidden, as loads from a single thread of execution cannot appear to be executed out-of-order. On ARM however, this behaviour is observable. Threads can "snoop" values from other threads' local caches without being compelled to observe all of the stores available in that cache. This allows stores to appear to "overtake" one another in the memory system – where a snoop into an adjacent cache can return faster load from main memory, allowing the weak execution of the MP litmus test.

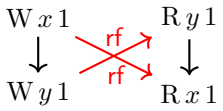
To better explain executions and weak behaviours in memory consistency models it is useful

Initially: $\texttt{x} = 0$

Thread 1	**Thread 2**
`x := 1;`	$\texttt{r}_1$ `:= x;`
`x := 2`	$\texttt{r}_2$ `:= x`

Figure 12: Pseudocode CoRR litmus test. Observing $\texttt{r}_1 = 2 \wedge \texttt{r}_2 = 1$ is forbidden on ARM.

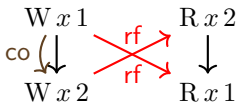
to draw executions graphically. First, let's examine the SC execution where $\texttt{r}_1 = 1 \wedge \texttt{r}_2 = 1$.



We will begin to use a style of drawing executions that is common in the literature (Batty et al. 2011; Lahav et al. 2017; Alglave, Maranget and Tautschnig 2014). Nodes indicate whether a value is read or written to memory, the location of the operation, and the value loaded/stored. For example, R x 0 indicates the value 0 is loaded from x, and W y 1 indicates that 1 is stored to y. Black arrows indicate *program order* (po), i.e. the order that the operations occur syntactically in the program. Red arrows indicate where a load read its value from, with the *reads from* (rf) relation.

Figure 12 shows an example to understand coherence order (co): the Coherence-ordered Read Read (CoRR) test shows us that for two reads in a row - observing the write of 2 hides the initialisation write. Coherence order is a total order[2] per-location, giving a "store order" to each variable.

The forbidden execution of CoRR is drawn below:

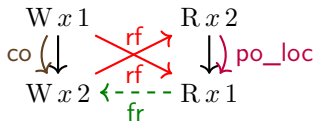


To understand why this execution is forbidden in the ARM memory model, we must look at a relation which can be calculated from rf and co. From-reads (fr) places reads to a given location into an order with respect to coherence order.

$$\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$$

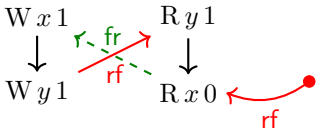
[2] co is a total order in multicopy-atomic (MCA) systems, as opposed to non-multicopy-atomic (non-MCA) systems where it is a partial order. Earlier revisions of the ARM ISA allowed non-MCA stores, but this has been revised and ARM is now MCA (Pulte et al. 2018).

Where $[-]^{-1}$ inverts a relation (reverses all the arrows), and ; composes relations (links arrows together). We can annotate the CoRR execution with an fr edge to demonstrate this. Here we can compose rf and co such that $(\mathrm{R}\,x\,1 \xrightarrow{\mathsf{rf}^{-1}} \mathrm{W}\,x\,1 \xrightarrow{\mathsf{co}} \mathrm{W}\,x\,2)$ forms an fr edge. Another derived relation is program order per location (po_loc) which is the subset of program order relating events on the same memory address. In this program with only one address all the po edges are also po_loc. The execution is drawn again below including the fr edge, and one of the po_loc edges highlighted.



An axiom included in the ARM memory model forbids cycles in po_loc ∪ fr ∪ rf ∪ co, making this execution forbidden as there the cycle $(\mathrm{R}\,x\,2 \xrightarrow{\mathsf{po\_loc}} \mathrm{R}\,x\,1 \xrightarrow{\mathsf{fr}} \mathrm{W}\,x\,2 \xrightarrow{\mathsf{rf}} \mathrm{R}\,x\,2)$ is present (ARM 2020; Deacon and Alglave 2019).

Now, recall MP (Figure 11) and consider the non-SC execution where $\mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 0$.



In this weak execution of MP, the read of x as 0 can be seen to come before the write of 1 to x. The rf edge drawn from a red dot it indicates the load reads from the initialisation. The implicit initialisation event is co-before all writes, which is what gives us the fr edge: $(\mathrm{R}\,x\,0 \xrightarrow{\mathsf{rf}^{-1}} \text{(initialisation)} \xrightarrow{\mathsf{co}} \mathrm{W}\,x\,1)$. Although this execution is similar in shape to CoRR, the absence of po_loc makes all the difference. The axiom which forbids the apparent reordering of reads to the same location does not apply here, the cycle is broken as there is no po_loc edge. Under x86 this execution is forbidden, writes cannot appear to overtake one another, similarly for reads. A key insight about ARM, then, is that the apparent re-ordering of loads or stores is observable, so long as they are not loads and stores to the same location.

Message passing is a useful communication primitive in software systems, so it is desirable to restore sequentially consistent behaviour. To do this, the ARM architecture provides barriers which prevent hardware re-ordering of loads and stores. The DMB (known as the Data Memory Barrier) instruction prevents instructions appearing to be executed out-of-order.

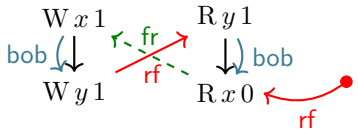
> *Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the* `DMB` *instruction are observed before any explicit memory accesses that appear in program order after the* `DMB` *instruction. It does not affect the ordering of any other instructions executing on the processor.*
>
> — armasm User Guide (ARM 2016)

Thus, to prevent the reordering of loads and stores and hence prevent weak execution of MP: a `DMB` instruction must be placed in each thread, as shown below.

Initially: $\mathtt{x} = \mathtt{y} = 0$

Thread 1	**Thread 2**
$\mathtt{x} := 1;$	$\mathtt{r}_1 := \mathtt{y};$
`DMB;`	`DMB;`
$\mathtt{y} := 1$	$\mathtt{r}_2 := \mathtt{x}$

Instructions separated by a `DMB` instruction are related by the *Barrier Ordered Before* (bob) relation in the corresponding executions (ARM 2020; Deacon and Alglave 2019), the weak execution of MP is drawn below, with the bob annotations.



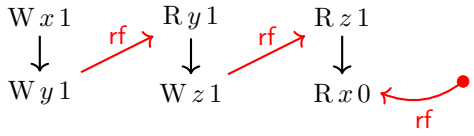
The ARM models forbids cycles in $(\mathsf{rf} \cup \mathsf{fr} \cup \mathsf{bob})$ so this execution is forbidden (Deacon and Alglave 2019; ARM 2020).

ISA2. An important concept in the ARM memory model is the cumulativity of synchronisation. When passing a message through multiple threads of execution it is important that synchronisation can be propagated. This concept demonstrated concisely with the ISA2 litmus test.

Initially: $\mathtt{x} = \mathtt{y} = \mathtt{z} = 0$

Thread 1	**Thread 2**	**Thread 3**
$\mathtt{x} := 1;$	$\mathtt{r}_1 := \mathtt{y};$	$\mathtt{r}_2 := \mathtt{z};$
$\mathtt{y} := 1$	$\mathtt{z} := 1$	$\mathtt{r}_3 := \mathtt{x}$

Observing $\mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 1 \wedge \mathtt{r}_3 = 0$ indicates that despite a flag ($\mathtt{y}$ and $\mathtt{z}$) being propagated between the 3 threads the data ($\mathtt{x}$) is not. This execution is permitted on ARM. The hardware will not enforce the order of the loads or stores.

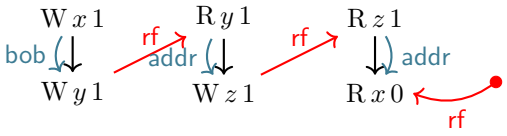


To rectify this, ordering must be enforced. For thread 1, the same `DMB` instruction can be issued to ensure ordering between the store of `x` and `y`. For threads 2 and 3, load-store ordering can be enforced using address dependencies. An operation A is address-dependent on an operation B when the value returned by operation B is required to compute the address used in operation A. A modified ISA2 with `DMB` and address dependencies is displayed below.

Initially: $\mathtt{x} = \mathtt{y} = 0$

Thread 1	**Thread 2**	**Thread 3**
`x := 1;`		
`DMB;`	$\mathtt{r}_1 := \mathtt{y};$	$\mathtt{r}_2 := \mathtt{z};$
`y := 1`	$*(\&\mathtt{z} + \mathtt{r}_1 - \mathtt{r}_1) := 1$	$\mathtt{r}_3 := *(\&\mathtt{x} + \mathtt{r}_2 - \mathtt{r}_2)$

The pointer arithmetic in the modified ISA2 has no effect on ultimate addresses, but it forces the hardware to preserve ordering between the load and store in thread 2, and the loads in thread 3. The address dependencies are represented by the addr relation. When we consider the weak execution of our modified ISA2 we find a cycle in (bob ∪ addr ∪ fr), which is also forbidden in the ARM memory model.

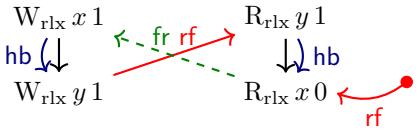


2.1.4 C++

The C and C++ Programming languages are critical low-level systems languages used in the majority of computer software. Windows is written in C++; Linux, BSD, and macOS are written in C (Torvalds 2021; Group 1983; Apple Inc. 2021); web browsers like Firefox and Google Chrome are written in C++ (Mozilla Project 2021; Project 2021). With C and C++ forming the foundations of critical computer software it is important that the languages are well understood and can be formally modelled. In the past the C and C++ standards were written carefully in English language prose, but had no formal mathematics behind them.

```
// Thread 1:
x.store_explicit(1, relaxed);
y.store_explicit(1, relaxed);

// Thread 2:
r1 = y.load_explicit(relaxed);
r2 = x.load_explicit(relaxed);
```




```
// Thread 1:
x.store_explicit(1, relaxed);
y.store_explicit(1, release);

// Thread 2:
r1 = y.load_explicit(acquire);
r2 = x.load_explicit(relaxed);
```

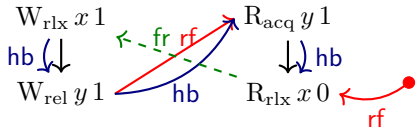


Figure 13: Two litmus tests in abbreviated C++ syntax. Relaxed message passing (MP) is on the left and release/acquire message passing (MP+rel+acq) is on the right. In both programs `x = y = 0` initially.

A requirement of the C and C++ programming language is that they give programmers direct access to the high performance code paths available in modern CPUs, like relaxed atomics and high performance vector instructions. In order to support this, the language memory model must permit the weak behaviours observable in the hardware, so the C and C++ memory models are weaker than their hardware counterparts (like x86-TSO and the ARM memory model). Were the C++ model to provide stronger guarantees it would mean that compilation of certain patterns of loads and stores would incur additional fences. For example, if C++ were designed to be as strong as x86-TSO the C++ ARM compiler would need to insert `DMB` barriers between each store instruction to prevent the weak MP behaviours observable on ARM hardware.

Happens before. With Mathematizing C++ (Batty et al. 2011) introduced a formal mathematical model of C++ which has been iterated upon and incorporated into the C++ standards text (ISO/IEC 2010). Since the adoption of the Mathematizing approach in the 2011 C++ standard, revisions have been made: most notably revisions proposed by Lahav et al. (2017) (known as RC11), which fixes the treatment of SC atomics in C++ have been adopted, but the RC11 approach to forbidding thin-air executions has not – this will be discussed in more detail in Section 2.3.1. The model is defined in terms of a "happens before" relation (hb), where cycles are forbidden. To explore this model, we will again look at a message passing (MP) litmus test. Figure 13 shows two versions of MP with different annotations to denote *access mode*. Access modes result in the compiler, one with a release/acquire pair and one with all relaxed accesses. The same $\mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 0$ outcome is drawn for each, but in the release acquire version the hb edges form a cycle, so the execution is forbidden by the C++ memory model. In the relaxed variant of MP we see that all the program order edges are in hb order, but the rf edge

is not. Accesses which appear in happens before order are synchronised, and the hb order of these actions can be relied upon in other threads of execution. In this way, we build lock-free racy programs with defined semantics – observable values are restricted by the actions available hb-before a given read.

To understand why, we can look at the definition of happens before, a subset of which is presented below.

$$\mathsf{hb} \subseteq ([E^{\sqsupseteq rel}]; \mathsf{rf}; [E^{\sqsupseteq acq}]) \cup \mathsf{po}$$

We can see that happens before contains all of program order (po) as well as rf edges which go between events of at least release strength to events of at least acquire strength. In both of the examples in Figure 13 we can see that the po edges coincide with an hb edge. In our relaxed program the rf edge between the load and the store are both relaxed, so the rf edge is not included in hb. Conversely in the release/acquire variant the rf edge goes between a store with acquire strength to a load with release strength, making the rf edge part of hb.

Another edge visible on these diagrams is the from-reads (fr) edge, also known as "reads before". fr indicates that a read observes some value which was written before some other write to the same location. In these weak MP executions the reads on `x` both read from the initialisation, which are coherence ordered before the store of `x := 1`. Along with rf and co, fr is among a category of relations which order operations on the same location. These relations are collected into one definition called *extended coherence order* (eco).

$$\mathsf{eco} \triangleq (\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})^+$$

C and C++ have a coherence axiom which states that $\mathsf{hb}; \mathsf{eco}^?$ must be irreflexive, preventing cycles in calculation of a value at one location.

$$\text{irreflexive}(\mathsf{hb}; \mathsf{eco}^?) \qquad \text{(C11 Coherence)}$$

This axiom forbids the weak execution of release/acquire MP – there is a cycle of $\mathsf{hb}; \mathsf{hb}; \mathsf{hb}; \mathsf{eco}$ –but permits the weak execution of relaxed MP as the rf arrow is not part of hb, breaking the cycle.

Races and Catch-fire semantics. Data-races are a programming hazard in C and C++, but building racy code can be useful in some circumstances. For example, when building a fast scheduler it is not critical that a task is selected from a queue deterministically. However, it is useful to be able to optimise code assuming that no code is racy, it allows reordering of

loads and stores which in practice yields significant execution speed-up. To allow these opposed pressures to coexist in C and C++ the standard has two flavours of non-synchronising memory access: non-atomic and relaxed. Races between relaxed accesses have defined semantics, such as that of the weak execution of relaxed MP shown on left in Figure 13 – these races appear in hb-order. Races between non-atomic accesses however exhibit *undefined behaviour* (UB): the whole program may do anything when code with UB is executed, including simply exiting with a 0 return code. For this reason, the C and C++ concurrency is known as a *catch-fire* semantics, where hitting UB means the program can have any execution at all, including totally nonsensical ones – colloquially the machine halting and catching fire is a valid execution for a racy program.

2.1.5 NVIDIA PTX

Modern graphics cards are have many thousands of threads accessing the same pool of shared memory, and the organisation of these threads is in part determined by the programmer and in part determined by the hardware.

NVIDIA provide a virtual instruction set for their graphics cards (Lustig et al. 2017). This instruction set acts as an intermediate between the assembly language for a given hardware generation and compilers, allowing for flexibility of implementers to change the instruction set between generations.

The PTX memory model is quite complex, as not only does it have an acyclicity relation along the lines of C++'s hb, but it is also *non-multicopy atomic* – meaning that two threads may observe stores to a single location in contradictory orders. The hardware gives programmers the facility to restore multicopy atomicity using special annotations, but those annotations require careful reflection of the program execution structure on the hardware.

2.2 Tools for evaluating memory models

As memory models are complex, and interactions between definitions are hard to exactly characterise there has been work on making automated tools for evaluating litmus tests under a given memory model. The most successful such tool is Herd7 (Alglave, Maranget and Tautschnig 2014), which provides a syntax for specifying axiomatic memory models as well as accepting tests generated by the LISA tool in LISA syntax. Other tools have been made to automatically compare memory models to discover litmus tests which tell them apart (Wickerson et al. 2017). Further tools can run litmus tests on hardware and characterise their behaviours. These tools have been important to discover previously undocumented hardware behaviours, particularly

```
SC

include "fences.cat"
include "cos.cat"

(* Atomic *)
empty rmw & (fre;coe) as atom

(* Sequential consistency *)
show sm\id as si
acyclic po | ((fr | rf | co);sm) as sc
```

Figure 14: CAT specification of a sequential consistency memory model

where hardware implementers have been reticent to work with academics.

2.2.1 Herd7

Herd7 (Alglave, Maranget and Tautschnig 2014) is a tool which allows memory model designers to express axiomatic memory models in a customisable language and to then automatically validate litmus test outcomes. It works in the `diy` suite of tools which provide a full suite of programs for litmus test synthesis, execution on real hardware, and mechanical means to determine which outcomes are permitted for a given litmus test according to a given memory model.

Herd provides a simple language for expressing memory models called the CAT language which has been formally specified by Alglave, Cousot and Maranget (2016). The language provides primitives for developing axiomatic memory models conforming to a broadly applicable interface. For example, (Alglave, Maranget and Tautschnig 2014) provides an example of a model of sequential consistency, printed in Figure 14.

This model defines two axioms which must hold for an SC execution of a program. The first is `empty rmw & (fre;coe) as atom` which specifies that the intersection of rmw and $\mathsf{fr}_e;\mathsf{co}_e$ is empty. Digesting that definition a little: it prevents execution fragments of the shape presented in Figure 15. In this execution fragment we can see a write apparently happening in the middle of an atomic read-modify-write operation. This "splitting the atomic" behaviour is forbidden by atomicity axiom of the Herd model above.

This model used several of Herd's built in features. Default definitions of relations like co, fr, rmw, as well as variants like external coherence (co_e) and external from-reads (fr_e). Similarly, there are built in axiomatic constraints like `empty` and `acyclic`. Herd therefore not only provides a framework to build memory models, but also a set of assumptions about the structure of the model. These assumptions have proven in general to yield sensible models, notably ARM's

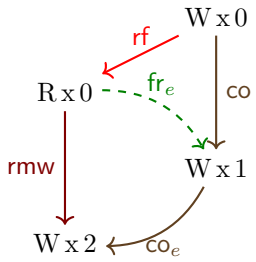


Figure 15: A fragment of an execution which would be forbidden by the atomicity axiom Herd SC model.

industrial model is a Herd model (Deacon and Alglave 2019), and C11 has a Herd model too. For some however, these assumptions prove to a be a little too restrictive. We will explore the NVIDIA PTX memory model in this thesis, which was originally expressed as a Herd model, but has since been re-expressed in a more custom Alloy-based approach.

2.2.2 `cppmem`

Alongside Mathemtizing C++ came a tool for automatically exploring the executions permitted by the proposed C++ memory model. The tool allows users to input tests in a C-like syntax and explore all the executions allowed by the model, as well as giving options to toggle some axioms of the memory model. Users can toggle certain axioms to see which axioms are giving the model strength against particular litmus tests. `cppmem` was essential in gaining acceptance of the new C++ model industrially, as it allowed the ISO C++ committee members to explore programs of interest and understand how the new memory model would affect them. Figure 16 shows a screenshot of `cppmem` running.

2.2.3 Alloy

Alloy is a relational model checking tool which has been used in several memory model formalisations. Unlike Herd, Alloy is a general purpose tool – offering no direct help with construction of memory models. However, Alloy being general purpose allows a memory model author to express more bespoke patterns, which has proved especially useful for memory models that describe graphics systems. Axioms can be expressed as facts over the relations of the model, and a model checker can be used to generate consistent executions within these relations and axioms. Predicates can be expressed to attempt to search for programs which exhibit some weak behaviour, and the model checker can search for counter-examples. Alloy has been used to define

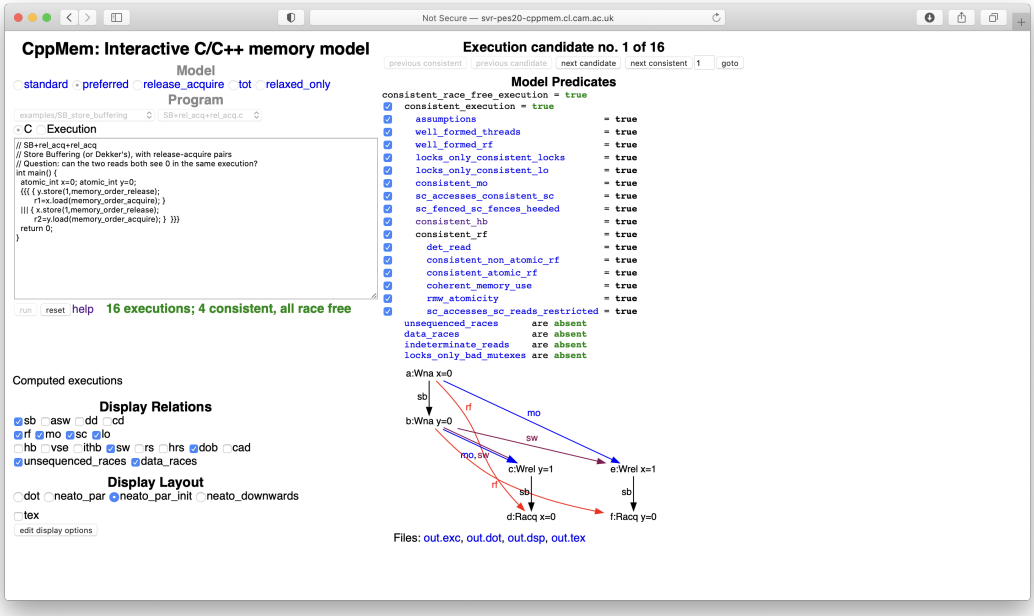


Figure 16: A screenshot of `cppmem` showing an allowed weak execution of the Store Buffering litmus test.

the PTX memory model (Lustig et al. 2017) (discussed in great detail in Chapter 3) and the Vulkan memory model (Khronos Group 2020), a cross-platform graphics API.

2.2.4 `memalloy`

`memalloy` is a tool for finding minimal litmus tests which can *distinguish* two memory models (Wickerson et al. 2017). Memory models can be distinguished when a program P executed under memory model M has an outcome which is permitted, but forbidden when executed under a different memory model M'.

$$\text{distinguishes}(P) \triangleq [\![P]\!]_M \neq [\![P]\!]_{M'}$$

`memalloy` existentially quantifies a P in a bounded universe to satisfy the constraint. The use of this is two-fold: it allows practitioners to demonstrate tests which can abstract complex axiomatic memory model details to communicate an intuition about the difference between models; and it enables the generation of new minimal litmus tests for automated testing. On a more meta-theoretic front, it provides a bounded means to test for strengthening or weakening of a model. When building a programming language model the language model should ideally

Initially: x := 0; y := 0;	
r_1 := y;	r_2 := x;
x := 1	y := 1

Figure 17: Load buffering litmus. $r_1 = r_2 = 1$ is allowed in C++.

be weaker than the architecture model, or specific modifications (such as inserting a fence) to simply generated code must be made to preserve semantic meaning. Relaxed atomics in C/C++ are specified weakly enough that they may be compiled to native load and store instructions on x86 and ARM with no additional memory operations, as the programming language model of relaxed atomics is weaker than the hardware model of loads and stores[3].

2.3 The Thin-Air Problem

Unfortunately axiomatic memory models have been found to be inadequate for specifying programming languages (Batty et al. 2015). The interaction between compiler optimisations and apparent program dependencies breaks the ability of an axiomatic model to use the source program to accurately describe its behaviour. Axiomatic models are either too strong, and forbid optimisations which are important in practice, or too weak and permit nonsensical program behaviours which are not observable on real hardware and also prevent verification of program components. This problem is known as the Out of Thin Air (OOTA) Problem. To explore OOTA the sensible place to start is with the Load Buffering (LB) litmus test, listed in Figure 17.

In the LB litmus test data is loaded from a shared variable, and then stored to a different variable. This specific example has a constant store of 1 to x and y, which does not depend on the values read into registers r_1 or r_2. In this instance, the execution where $r_1 = r_2 = 1$ is allowed by ARM, and as such is also allowed in C++ – forbidding this execution in C++ would necessitate some compilation strategy which prevented the behaviour when targetting ARM, like emitting an additional fence in each thread.

In Figure 18 the program has been modified to store values to x and y which are data-dependent on the value loaded into r_1 and r_2. This program is known as LB+datas, and the outcome where $r_1 = r_2 = 42$ would be quite unreasonable: to store 42 to x we must first have loaded 42 from y; to store 42 to y we must have first loaded 42 from x. There is no interleaving or re-ordering of instructions which could yield this result, the calculation of the value for 42 is

[3]There is a caveat for mixed-sized atomics which are poorly specified for hardware models (Flur et al. 2017)

Initially: `x := 0; y := 0;`

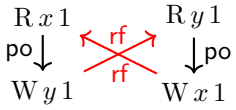
r_1 := y;	‖	r_2 := x;
x := r_1	‖	y := r_2

Figure 18: Load buffering litmus test with data dependencies. The thin-air behaviour would be observing that $r_1 = r_2 = 42$.

self-referential. Informally, an execution where this outcome is observed would require some cycle in dependency (dep) and rf. We might consider, then, a rule which forbids cycles in (dep ∪ rf). Next, consider a small variation of this test, where the data dependencies are replaced with control dependencies.

Initially: `x := 0; y := 0;`

r_1 := x;	‖	r_2 := y;
if($r_1 = 1$){y := 1}	‖	**if**($r_2 = 1$){x := 1}



acyclic(po ∪ rf)

The test starts with x and y initialised to 0, then two threads concurrently read and conditionally write 1 back to their respective variables. The outcome $r_1 = r_2 = 1$ (1/1) is unintuitive, and it cannot result from SC: there is no interleaving that agrees with the program order and places the writes of 1 before the reads for both x and y.

In an axiomatic specification, the outcome specified by the test corresponds to the execution graph shown alongside the source code. The axiom acyclic(po ∪ rf) forbids the outcome 1/1 as the corresponding execution contains a cycle. Indeed the SC, x86, Power and ARM memory models each include a variant of this axiom, all forbidding 1/1, whereas the C++ standard omits it (Batty et al. 2011) and allows 1/1.

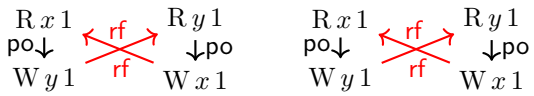
To better understand why C++ does not forbid the cycle in po ∪ rf, we will look at a further example which illustrates why axiomatic models are not well suited for programming language memory models.

Axiomatic specifications do not fit optimised languages. Languages like C++ and Java perform dependency-removing optimisations that complicate their memory specifications. For example, the second thread of the LB+false-dep test (presented below) can be optimised using common sub-expression elimination to $reg2$ `:= y; x :=` 1. This kind of control dependency is said to be a *false dependency*, as the outcome of the if-statement does not rely on the value read in $\texttt{r}_2$: it will write `x :=` 1 in the true and false branches. In the unoptimised code, the control dependency would have prevented code motion. However in the compiled code where the compiler optimises this control flow away, ARM and Power allow load/store reordered where there is no dependency between the operations, so the relaxed 1/1 outcome is allowed.

Initially: `x :=` 0`; y :=` 0`;`

$\texttt{r}_1$ `:= x;`	‖	$\texttt{r}_2$ `:= y;`
if($\texttt{r}_1 = 1$)`{y := 1}`		**if**($\texttt{r}_2 = 1$)`{x := 1}` **else** `{x := 1}`

The memory model of the C++ standard (ISO/IEC 2010) is flawed because its axiomatic model cannot draw a distinction between the executions leading to outcome 1/1 in LB+ctrl and LB+false-dep. To see that the dependency is false, one must consider more than one execution path, but axiomatic models judge single executions only (Batty et al. 2015). This is illustrated below, where the 1/1 executions of LB+false-dep and LB+ctrl are drawn. The executions are identical with no distinguishing features, as all control flow choices are elided once programs are reduced to executions.



2.3.1 RC11

Lahav et al. (2017) have provided a simple solution to the out-of-thin-air problem, in their paper addressing some issues with the treatment of SC atomics in the 2011 C and C++ standards. The tweak is to introduce a new axiom which simply forbids cycles in $\mathsf{rf} \cup \mathsf{sb}$, ($\mathsf{sb}$ is equivalent to po in most situations), which conveniently forbids the problematic LB+false-dep executions.

Unfortunately it would also forbid executions of plain load buffering (as in Figure 17), where today important compiler optimisations would remove dependencies or permit hardware reordering.

This solution is appealing if important hardware and compiler optimisations didn't introduce load/store reordering. It also has the appeal of being only a minor change to the ISO C++ standards text, which would be simple to explain to programmers. However there are hardware targets that do allow load/store re-ordering, and RC11 would impose new restrictions on C++ compilation to those targets. The C++ compiler would need to insert barriers or fences to prevent code motion between all loads and stores, which in turn would create a performance penalty on those platforms. For x86 and ARM, Ou and Demsky (2018) showed that the cost of forbidding load/store reordering is small in most cases, but more research is needed to understand the performance implications on other hardware such as POWER or NVIDIA's graphics processors.

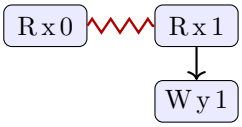
2.3.2 Jeffrey and Riely

Jeffrey and Riely (2016) (J+R) presented a novel approach to Java concurrency semantics with their Event Structures model. This model uses event structures to encode all possible executions of a program, and then a game semantic type model to explore the executions of the model. To "win" the game, and find an execution as allowed one must justify each write while an opponent can find control flow choices to attempt to confound justification of events required to progress to the execution of interest. Event structures are popular in several weak memory models, including the model which will later be presented in Chapter 5. To understand event structures better, it is useful to interpret a few programs into event structures. Event structures show an overlay of all possible executions of a program, by generating a read event for each value a load could return. For a large set of values, this is obviously unruly – so we normally consider a small subset of values which are large enough to show interesting behaviour without being impractical to draw or compute over. Let us first interpret a read of `x` using a value set of $\{0, 1\}$.

This diagram shows the two read events, and draws a *conflict* (∿) edge between them. Conflict indicates where two events cannot occur in the same execution – this is where different outcomes can result from different dynamic executions of the program. If we now consider how a read might be composed with a larger program, we can see how the control flow of a program can be encoded in an event structure.

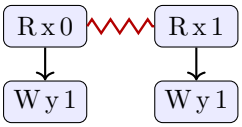
```
r1 := x;
if(r1 = 1){
   y := 1
}
```

Using the same value set, this is interpreted into the following event structure:



Here we see that a new write event is added to the end of the event structure, but only under the R x 1 event. Conversely, in the case we have a false dependency, we see something different:

```
r1 := x;
if(r1 = 1) {
   y := 1
} else {
   y := 1
}
```



In this example, there is a W y 1 in both branches of the event structure. Like our original program, the false dependency is evident. The J+R memory model seeks out these symmetries across conflicted branches of execution to find reads that are always justified. This exploration is phrased as a ∃∀∃ formula over event structures which, informally, encodes a 2-player game. For some chosen execution (∃), no matter what move an opponent makes (∀), the player should be able to find a counter move which allows their configuration to be justified (∃). The alternation of quantification is important: it makes the semantics higher-order, which means it cannot be solved with first-order SAT solvers without a substantial penalty for converting the higher-order function into a first order function. This observation leads to the work in Chapter 4 where we use a higher-order model checker to automatically evaluate the J+R memory model over litmus tests.

2.3.3 A Promising Semantics

Kang et al. (2017) present a detailed operational model where one can "promise" a value will be written to a variable and may be observed by another other thread, if one can prove that promise

$$\textsc{Thread Step}\ \frac{\sigma \xrightarrow{\tau} \sigma' \quad \mathcal{TS} \mapsto \mathcal{TS}' \quad \mathcal{M} \mapsto \mathcal{M}' \quad \ldots}{\langle \mathcal{TS}, \mathcal{M} \rangle \to \langle \mathcal{TS}', \mathcal{M}' \rangle} \qquad \textsc{Internal}\ \frac{\mathcal{TS} \mapsto \mathcal{TS}' \quad \mathcal{M} \mapsto \mathcal{M}' \quad \ldots}{\langle \mathcal{TS}, \mathcal{M} \rangle \to \langle \mathcal{TS}', \mathcal{M}' \rangle}$$

$$\textsc{Machine Step}\ \frac{\begin{array}{c}\langle \mathcal{TS}(i), \mathcal{M} \rangle \to^* \langle \mathcal{TS}', \mathcal{M}' \rangle \\ \langle \mathcal{TS}', \mathcal{M}' \rangle \xrightarrow{e} \langle \mathcal{TS}'', \mathcal{M}'' \rangle \\ \langle \mathcal{TS}'', \mathcal{M}'' \rangle \text{ is consistent}\end{array}}{\langle \mathcal{TS}, \mathcal{M} \rangle \xrightarrow{e} \langle \mathcal{TS}[i \mapsto \mathcal{TS}''], \mathcal{M}'' \rangle}$$

Figure 19: A simple structural view of The Promising Semantics, with the state re-write rules elided.

can be satisfied by some future write of the program. The Promising model works in a similar way to processor speculation, where local execution of a thread can proceed under the assumption that a load will return a particular value. The operational model is structured as a nested transition system where the machine may take Machine Steps, Thread Steps or perform a Certification. Taking each in turn: one selects a thread to take a Machine Step, where a single thread of the program may make some number of Thread Steps. Thread Steps may make new promises and fulfil existing ones, or make Internal transitions. When the sequence of Thread Step transitions is complete, there may be outstanding promises at the end of the outer Machine Step. The operational model requires a proof that each of these outstanding promises can be fulfilled by checking Certification – the "is consistent" check in the Machine Step rule. Certification involves finding some future sequence of Thread Steps which can be made to clear the promise set. A high-level view of this structure is presented in Figure 19, showing only the transition system and not the complete set of rules from Promising.

This nesting of transitions creates a nesting of quantification (and alternation of quantification) that begins to reflect the game semantics of J+R, where for any choice of read all futures must have some way of justifying that read. This deeply nested transition system is problematic for implementing a tool. At each step through interpreting a program a combinatorial choice of possibilities is produced: when to make promises, when to fulfil, when to end a thread step and make a new machine step, which thread to execute in the next machine step…

For this reason Promising is a dissatisfying solution to the Out-of-thin-air problem, it neither digests a program to some core meaning as we do in MRD (see Chapter 5), nor does it directly express the structure of execution exploration as lucidly as J+R have. Further, because Promising is so complex to execute, it is unreasonable to build a complete tool for exploring the state transition system that Promising presents – so validation of the model can only be done abstractly

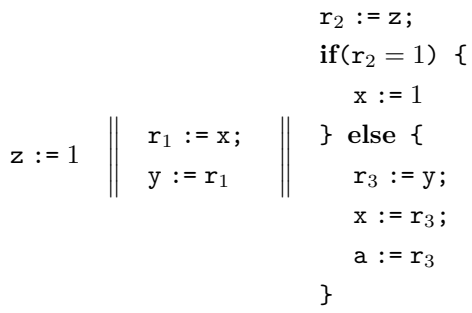


Figure 20: The thin-air execution where 1 is written to `a` is observable under the Promising Semantics.

rather than by exhaustive testing. Indeed, the authors of Promising do provide some formal verification of their semantics and do not present a tool. There is no proof of decidability for the Promising model as presented, only weakened versions, such as that of Abdulla et al. (2020). Abdulla et al. (2020) further show that only with the restriction of bounded promises in a relaxed-only subset of Promising can decidability be restored.

A more serious problem for the Promising semantics is that it allows a thin-air execution which breaks type safety. Jagadeesan, Jeffrey and Riely (2020) provide an example based on a type safety litmus test for Java (Lochbihler 2014), printed in Figure 20. This behaviour is thin-air, as any execution where $\mathtt{x} = 1$ must have been from a context where $\mathtt{r}_2 = 1$ – there is cyclic dependency to get to an execution where `a` is written as 1. Promising allows this because the promise mechanism allows $\mathtt{x} = 1$ to be speculated using certification from one branch of the conditional, other threads to make progress using that promise ($\mathtt{r}_1 = \mathtt{x};\ \mathtt{y} = \mathtt{r}_1$), and then for the promise to be fulfilled by the false branch of the conditional in a future Machine Step. Indeed, Jagadeesan, Jeffrey and Riely (2020) confirm that Promising allows this thin-air execution with the authors of Promising, and conclude that Promising "cannot support both type safety and realistic memory reclamation".

2.3.4 WeakestMO

Chakraborty and Vafeiadis (2019) presented another event structures based model, where "consistent" event structures are built and from those "consistent" executions are then projected. In this paper, Chakraborty and Vafeiadis (2019) observe that the Promising Semantics (Kang et al. 2017) admits a problematic execution to a program they call Coh-CYC.

```
x := 2;                          ||   x := 1;
r1 := x; // 3                    ||   r2 := x; // 2
if(r1 != 2){y := 1}              ||   r3 := y; // 1
                                 ||   if(r3 != 0){x := 3}
```

In this program, Promising allows the execution where $\mathtt{r}_1 = 3 \wedge \mathtt{r}_2 = 2 \wedge \mathtt{r}_3 = 1$, corresponding to a either break down in per-location coherence order, or a circular dependency.

Promising allows this by fulfilling a promise made with an assumption about one thread of execution with a fulfilling write from a different thread of execution without checking any coherence constraints (Kang et al. 2017; Chakraborty and Vafeiadis 2019).

In Chakraborty and Vafeiadis (2019), two models are defined with similar construction: WEAKEST and WEAKESTMO. WEAKEST is a weaker event structures model which is designed to be weaker than Promising (Kang et al. 2017), even though that means admitting the problematic execution of Coh-CYC. As WEAKEST is weaker than Promising, this allows Chakraborty and Vafeiadis (2019) to show that their approach can yield a semantics which can be efficiently compiled to common hardware targets - reusing the proof of Kang et al. (2017). WEAKESTMO on the other hand is slightly stronger than WEAKEST, while being incomparable to Promising. WEAKESTMO forbids the problematic execution of Coh-CYC by maintaining a per-location modification order. It also retains a compilation correctness result, but efficient mappings cannot be used for POWER (Chakraborty and Vafeiadis 2019).

Chapter 3

Extending the PTX Memory Model

In this chapter we explore an extension to the memory model of NVIDIA's pseudo instruction set, PTX. PTX is a compiled assembly language which goes through light optimisation and translation to real hardware instructions. PTX allows NVIDIA to produce low level tools which are hardware agnostic, permitting them to make architectural changes to their hardware without causing intergenerational software incompatibility. We will extend the PTX memory model to cover architectural features which allow incoherent memory accesses through specialised graphics instructions. The loads and stores of these graphics operations are treated differently in the caches, and the behaviours exposed do not match with the existing PTX memory model. We extend the PTX memory model with a partial order on memory accesses, allowing accesses on the same location to be weakly ordered when the operations are of mixed type, e.g. a general purpose load of `x` followed by a texture load of `x`. The extended PTX model has been evaluated using an Alloy approach similar to that described in Section 2.2.4, as well with automated theorem proving techniques. Examining the PTX memory model sheds light on an interesting property of most hardware models, architectural (and in this case PTX compiler) optimisations do not remove dependencies. As a result, the PTX memory model has the simple restriction that $\mathit{acyclic}(\mathsf{rf} \cup \mathsf{po})$ – just as RC11 has (see §2.3.1) – meaning that Out-of-thin-air executions are neatly forbidden and do not complicate the memory model any further.

3.1 Introduction

NVIDIA PTX is the virtual Instruction Set Architecture (ISA) for NVIDIA graphics processors. These graphics processors can also be used as general purpose multiprocessors, and programmed in domain specific languages like CUDA or OpenCL which allow a programmer to build "single instruction multiple data" (SIMD) programs in a setting familiar to a C++ programmer. PTX is the language which is targeted by the CUDA compiler, and subsequently is compiled by the graphics driver into native code which can run on the GPU (NVIDIA Corporation 2019).

> "The PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers."

The abstraction afforded by the virtual instruction set allows NVIDIA leave PTX code as hardware agnostic, translating it to the native instruction set of any current or future hardware revision. In recent years, effort has been focussed on constraining the memory semantics of PTX, specifically giving it a memory consistency model (NVIDIA Corporation 2019). In order to accelerate both graphics and compute workloads, GPUs provide certain specially-annotated *surface*, *texture*, and *constant* memory accesses. The memory model has been used to describe the compute portion of PTX, but has not given any semantics to interactions between compute portions of the GPU architecture and the graphics components.

The specialised memory accesses provided for acceleration can touch data which may also be accessed with *generic* accesses. However, they have different memory semantics due to their being performed through a set of specialised caches optimised for each type of access. Figure 21 gives a high level view of this, where a single address space is shown at the top of the diagram, and multiple parallel caches can hold a copy of the same piece of data. This diagram will be explored in greater depth in Section 3.2.1. We say that memory operations which go via these named parallel caches are performed via *non-generic views of memory*. These caches are said to be *non-coherent*, that is to say operations visible in a given order in a generic cache might be visible in a different order in a non-generic cache.

In this chapter we will explore how to include the non-generic views of memory in the PTX memory model, and present evaluation on how these changes were validated.

3.2 Views

Mixed-view memory models aim to provide a better-defined, more general-purpose programming model for specialised caches found on today's CPUs, GPUs, and accelerators.

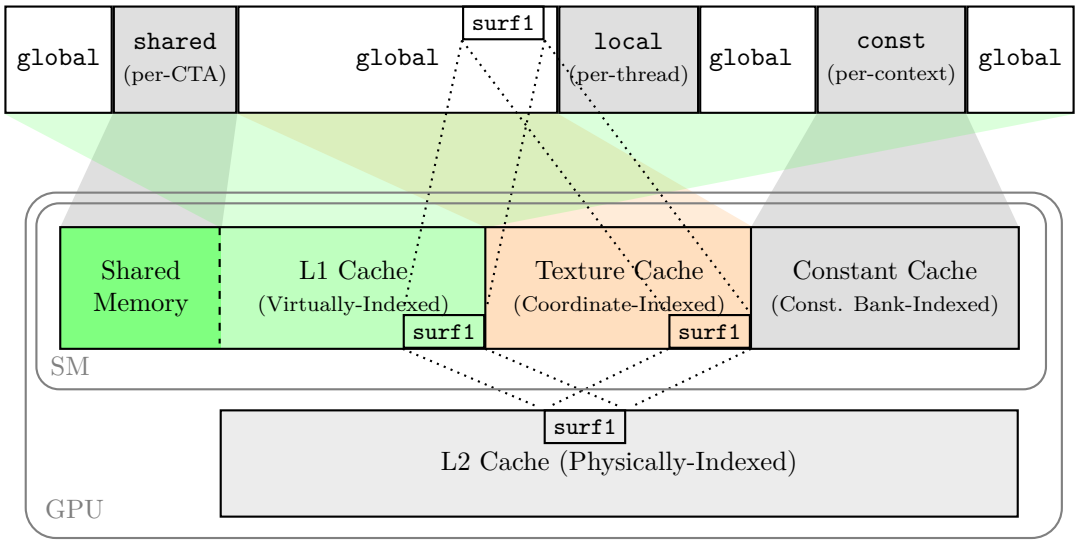


Figure 21: The memory space of a GPU showing the common address space, and multiple caches for the same data (as represented by `[surf1]`).

3.2.1 Background: GPU State Spaces and Specialised Caches

We start by explaining the motivation behind the state spaces and specialised caches used in NVIDIA GPUs. Although we follow NVIDIA terminology throughout this chapter, the concepts are generic. Figure 21 presents the overall picture, which we will describe in detail in this section.

The NVIDIA GPU programming model exposes five general-purpose *state spaces*, or address spaces: *global*, *local*, *shared*, *constant*, and *generic* memory. Global memory is the general-purpose read-write address space accessible by all threads in a program[1]. Shared memory is a special region of memory private to each cooperative thread array (CTA, i.e., thread block). Local memory is likewise a special region private to each individual thread. Constant memory optimises the performance of read-only data buffers. Generic memory is a superset address space: "[t]he state spaces `const`, `local` and `shared` are modeled as windows within the generic address space. [...] A generic address maps to `global` memory unless it falls within the window for `const`, `local`, or `shared` memory." –NVIDIA Corporation (2019) (§6.4.1.1). Generic pointers allow for a single piece of code to be used on global memory, shared memory, or local memory interchangeably.

Every memory access instruction in PTX, the virtual ISA for NVIDIA GPUs, either explicitly or implicitly indicates the *state space* (`.ss`) through which it is addressing memory. A summary is given in Figure 22. Normal loads, stores, and atomics make the choice explicit through the `.ss` modifier, with some combinations (e.g., storing to constant memory) disallowed by

[1]We avoid using the generic term "shared memory" to refer to global memory due to the conflict with NVIDIA's distinct use of the term.

Instruction(s)	Description
`ld.ss. st.ss`	Memory access to a state space `.ss`
`atom.ss`, `red.ss`	Atomic access to a state space `.ss`
`ld.global.nc`	Global load, cached in texture cache
`tex`, `tld`, `suld`	Texture/surface load via texture cache
`sust`, `sured`	Surface store/reduction texture cache

For ld, st, atom, and red:
`.ss={.const,.global,.local,.param,.shared};`

Figure 22: Basic instruction set summary

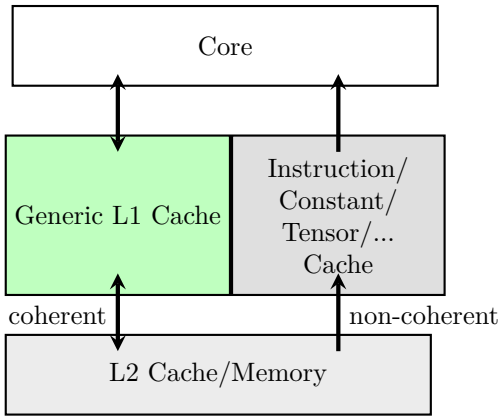


Figure 23: A more basic view-aware architecture

the architecture. Texture and surface instructions access the same address space, but through coordinates relative to resources described by opaque descriptor data structures.

Most global, local, and generic memory accesses are performed through the L1 cache of the streaming multiprocessor (SM) – the blocks processors that NVIDIA GPUs consist of. Other accesses are performed through separate data paths optimised for particular graphics use cases. Texture and surface accesses are cached in a specialised read-only *texture cache* that is addressed by texture addressing modes (e.g., coordinates) rather than "normal" memory addresses, as this speeds up texture processing. Constant loads are cached in a dedicated *constant cache* that speeds up accesses to constant memory. Shared memory is pinned in a dedicated portion of the L1 cache. All of these caches exist to provide better performance than the default data-path through the L1 cache. For example, shared memory accesses are guaranteed to perform approximately the same as an L1 cache hit, which provides roughly an order of magnitude benefit in latency and bandwidth over accessing the L2 cache.

3.2.2 Motivation: Using Specialised Processor Caches

Modern processors increasingly have fixed-function hardware with specialised caches in order to speed up certain types of operation. We model these specialised caches as *views* of memory, and consider a use case in existing production hardware. To help build intuition about views, one instance could be an instruction view: instruction caches are often incoherent with respect to memory, and many architectures require specialised instructions (e.g., `isb` on ARM, `isync` on Power) to synchronise instruction memory with data memory. This principle can easily be applied to any number of caches specialised for different purposes on CPUs, GPUs, and/or accelerators, ranging from architectures as simple as the one shown in Figure 23 to sophisticated modern GPUs like the one discussed in Section 3.2.

In addition to existing designs, we expect our mixed-view memory model to support even more heterogeneous and/or forward-looking designs with new caches. Suppose a future GPU architecture were to support read-write texture caches, as opposed to the read-only caches in existing NVIDIA GPUs. Some graphics APIs provide support for such operations within the context of a more restrictive graphics programming model (Khronos Group 2020; NVIDIA 2018), but both compute and graphics would benefit from being able to programmatically generate surface or texture resources on the fly. A similar but more widely-used example is self-modifying code on architectures with non-coherent instruction caches. The value of being able to programmatically synchronise instruction memory with data memory has proven tremendously useful in many just-in-time (JIT) compilation scenarios.

We also envision specialised caches enabling brand new features. For example, consider a scenario in which one grid launching another grid (NVIDIA 2014) wanted the ability to store to "`constant`" or `param` memory directly. Doing so would eliminate the latency of the round trip to the host CPU that is otherwise needed when setting up kernel arguments and constants (Section 3.2.1). Lastly, imagine that an accelerator optimising for machine-learning applications wanted to add a specialised cache indexed by tensor coordinates rather than memory addresses. Such a cache would behave very much like a texture cache on GPUs, and hence it would be naturally supported by the approach we propose.

3.3 Observing mixed view behaviour

It is possible to exercise views in CUDA programs today. Since all view caches are assumed to be backed by a common set of global addresses, it is possible to access the same region of memory through multiple distinct caches. Consider a video game designer that wishes to

```
#define N 256
__constant__ int const_array[N];
__global__ void kernel(int *global_array_ptr,
                       int *result) {
    global_array_ptr[0] = 42;
    __threadfence();
    /* Copy const_array to result using ld.const */
    for (int i = 0; i < N; i++)
        result[i] = const_array[i];
}
int main(int argc, char* argv[]) {
    /* ... */
    cudaGetSymbolAddress((void**)&global_pointer,
                         constant_array);
    kernel<<<1,1>>>(global_pointer, d_result);
}
```

(a) Original CUDA code

```
mov.u64         %rd7, global_array_ptr;
mov.u32         %r23, 42;
st.global.u32   [%rd7], %r23;
membar.gl;
mov.u64         %rd19, const_arr;
ld.const.u32    %r24, [%rd19];
```

(b) Relevant subset of the emitted PTX

Figure 24: If one writes to constant memory using a global memory pointer, i.e., by setting `global_array_ptr` equal to `const_array`, then the stored value may not be propagated to the subsequent constant load

programmatically generate new a graphics surface on the fly based on a player's action. The code to do this might write to the surface object `surf1` using a global store and then read the same object back using a surface load. As Figure 24 depicts, the store would be performed through the L1 cache, while the load would be performed via the texture cache. Unfortunately, none of the caches within the SM are kept coherent with each other or with the L2 cache (NVIDIA Corporation 2019; NVIDIA 2018). As a result, if `surf1` were already present in the texture cache, the load in this example would simply hit on that (now-stale) value, blissfully unaware that a prior store from the same thread had overwritten the address in question. This clearly violates the single-threaded memory semantics that programmers expect.

As another more concrete example, Figure 24 presents a CUDA program that attempts to store to a global address backing a region of constant memory during program execution. While a programmer might expect stores to `global_array[0]` to update the value in `const_array[0]`,

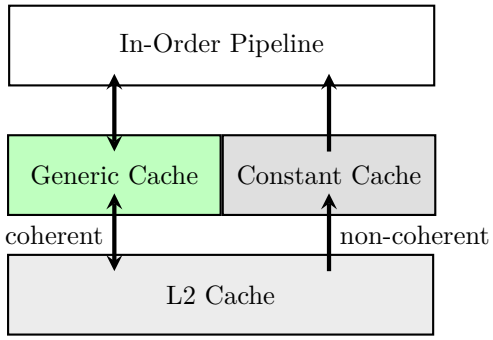


Figure 25: Hypothetical SC processor with constant cache

the new value 42 is not propagated to the incoherent constant cache, leaving the subsequent constant load of `const_array[0]` to continue reading the original value. We have observed this behaviour on existing NVIDIA hardware.

As a consequence of weak memory behaviours like those described above, the documentation simply states that "any texture fetch or surface read to an address that has been written to via a global or a surface write in the same kernel call returns undefined data" (§2.3), and that "[t]he memory consistency model does not apply to texture and surface accesses." (§8.1) NVIDIA Corporation (2019). It is exactly this gap in the memory model that we have addressed in this chapter.

Although it would be desirable to update constant memory in device, currently the only way to clear the constant cache is through a kernel launch, which has substantial latency. Our modification to the PTX memory model provides a description of hardware behaviours observable today, explaining that loads from constant memory may not be witness to updates via generic memory. Further, we provide semantics for operations which will clear the constant cache (such as a kernel launch, or a view fence).

3.4 Sequential Consistency with Constant Caches

To introduce the formalism of a mixed view memory model, we will first take a simple SC memory model and then extend that with views and view fences. Consider a basic processor that implements sequential consistency for normal data accesses, but accesses specially-allocated constants through a read-only non-coherent constant cache. Such an architecture would follow the model of Figure 25, using in-order pipelines and a constant view cache. The lack of coherence poses no problem if the constants are not modified, but the workload may need to set up new constants in order to enter a new phase of the program.

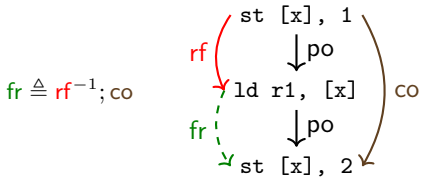
$$\mathsf{address} \subseteq (\Sigma \times \mathcal{A}) \quad \mathsf{po} \subseteq (\Sigma \times \Sigma) \quad \mathsf{rf} \subseteq (\Sigma_{\text{Write}} \times \Sigma_{\text{Read}})$$
$$\mathsf{co} \subseteq (\Sigma_{\text{Write}} \times \Sigma_{\text{Write}}) \qquad \mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$$

Figure 26: The formalised primitive relations of an SC memory model

3.4.1 Baseline Formalisation of SC

Our mixed-view SC model begins with the following primitive sets: $\mathcal{A}$ is the set of memory addresses, and Σ is the set of events. In this SC model, there are three types of events: Read, Write, and View Fence. The set of events restricted to a type t is denoted as Σ_t, e.g. Σ_{Write} is the set of write events. We define two basic relations: "address" relates each memory access to the address it accesses, and "program order" (po) is a total ordering of the events in each thread.

Although Lamport (1979) first defined sequential consistency in terms of a single total order over what we call Σ, modern axiomatic approaches favour the use of a more primitive set of relations, each of which corresponds to a behaviour that is more directly observable (Alglave, Maranget and Tautschnig 2014). We adopt the common relations and notation. The "reads-from" (rf) relation relates each store to all loads that return the value written by that store. In litmus test diagrams, an rf arrow drawn with no source indicates that a load returns the (implicit) initial value in memory for the memory location in question. The "coherence" (co) relation is a total order over all stores to the same address. The "from-reads" (fr) or "reads-before" relation relates each load to the co-successors of the store from which it returned its value. The "address" relations (address) relates each event to an address, it is used to find accesses on the same location – the relation $\mathsf{address}; \mathsf{address}^{-1}$ will relate all memory operations at the same location. With that definition we define "program-order per location" (po_loc) as the subset of po which relates accesses at the same location. All stores are considered co-successors of the implicitly-modelled store writing the initial value of each memory location. The relations are formalised in Figure 26, and an exemplary diagram showing fr is printed below.



In plain English, fr is defined as one rf edge followed in reverse, followed by a co edge, as shown

$$\mathsf{rfe} \triangleq \mathsf{rf} \setminus \mathsf{po} \setminus \mathsf{po}^{-1} \qquad \mathsf{ppo} \triangleq \mathsf{po} \setminus (\Sigma_{\text{Write}} \times \Sigma_{\text{Read}})$$
$$\mathsf{fence} \triangleq \mathsf{po} \cap ((\Sigma \times \Sigma_{\mathrm{F}_{SC}}) \cup (\Sigma_{\mathrm{F}_{SC}} \times \Sigma))$$
$$\mathit{acyclic}(\mathsf{rfe} \cup \mathsf{co} \cup \mathsf{fr} \cup \mathsf{ppo} \cup \mathsf{fence}) \qquad \text{(TSO)}$$
$$\mathit{acyclic}(\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr} \cup \mathsf{po\_loc}) \qquad \text{(SC per Location)}$$

Figure 27: Collected definitions for TSO.

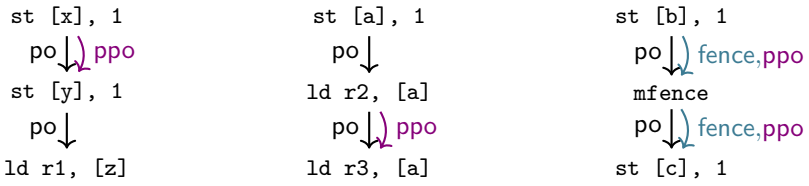
above. With this notation defined, the definition of sequential consistency becomes the following:

$$\mathit{acyclic}(\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr} \cup \mathsf{po}) \qquad \text{(SC)}$$

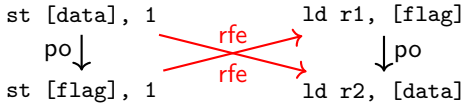
In other words, the acyclicity of this set of relations implies the existence of a total order over Σ; this is exactly the Lamport total order. The goal will be to incorporate our basic notion of constant cache synchronisation into this model.

3.4.2 Drawing Inspiration from TSO

To formalise mixed-view SC, we draw inspiration from the way SC may be weakened into total store order (TSO) (Sewell et al. 2010; Alglave, Maranget and Tautschnig 2014). TSO makes three changes to SC. First, TSO allows reads to bypass earlier writes. The TSO axiom therefore replaces po in the SC axiom with a new relation *preserved program order* (ppo) that maintains the subset of po not ordering a write before a read, unless a fence lies in between (fence).



Second, TSO allows loads to forward values from stores still in a thread-private store buffer. This has the effect of making it appear to other threads that the read has been performed before the write from which it returns a value. As such, the TSO axiom also replaces rf in the SC axiom with *reads from external* (rfe), which excludes intra-thread forwarding.



Lastly, to restore the expected single-threaded behaviour for intra-thread forwarding in spite of the apparent reordering, TSO requires a second axiom usually called *sequential consistency*

per location, where *location* is a synonym for address. The definitions of these relations and the axioms are collected in Figure 27.

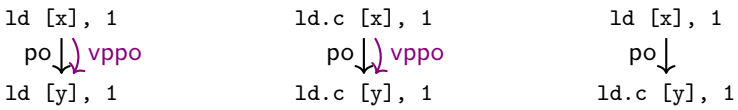
3.4.3 Formalising Mixed-View SC

Now, to formalise mixed-view SC, we take a similar approach to the weakening of SC into TSO. We need two new concepts. First, we define $\mathcal{V}$ to be the set of views in the model. In this case, there are two: the generic view (GEN) and the constant view (CONST). We denote the set of events with a view v using Σ^v, and relate events to their view with the view relation.

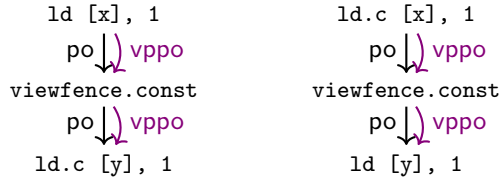
$$\mathcal{V} \triangleq \{\text{GEN}, \text{CONST}\}$$

We also define *view fences* ($\text{F}_\mathcal{V}$), which may be used by a programmer to restore SC ordering between different views and generic memory, the set of view fence events is written $\Sigma_{\text{F}_\mathcal{V}}$. Second, we define the notion of *view preserved program order*, or vppo. The effect of incoherent constant caches is that constant reads may hit on stale data, as if they had been performed as normal generic reads taking place earlier in program order. This means that po in the original SC axiom must be weakened. To match the behaviour of the architecture, we define vppo as the union of the following three components:

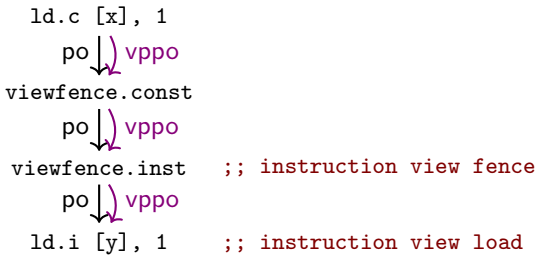
1. $\bigcup_{v\in\mathcal{V}}(\Sigma^v \times \Sigma^v)$: pairs of events in the same view are performed via the same cache, and therefore preserve normal po ordering.



2. $(\Sigma^{\text{GEN}} \times \Sigma_{\text{F}_\mathcal{V}}) \cup (\Sigma_{\text{F}_\mathcal{V}} \times \Sigma^{\text{GEN}})$: view fences ($\text{F}_\mathcal{V}$) enforce ordering between generic accesses and accesses to the associated view. Note in these examples the vppo edges from the `viewfence.const` to the `ld.c` are because of the previous rule, but this time there are also vppo edges between the generic `ld` event and the CONST view event `viewfence.const`.



3. $\Sigma_{\mathrm{F}_\mathcal{V}} \times \Sigma_{\mathrm{F}_\mathcal{V}}$: in models with more than one non-generic view, fences of different views can pair together to synchronise a view v_1 with a different view v_2 via an (implicit) intermediate hop of generic memory. The example below has an additional hypothetical instruction view.



Events of different views not separated by appropriate view fences are *not* preserved within program order, as they will be performed via a set of non-coherent caches, and may therefore violate normal single-threaded memory semantics. Putting the pieces together, we get:

$$\mathsf{vppo} \triangleq \bigcup_{v \in \mathcal{V}} \begin{pmatrix} (\Sigma^v \times \Sigma^v) \cup (\Sigma^{\mathrm{GEN}} \times \Sigma_{\mathrm{F}_\mathcal{V}}) \\ \cup\, (\Sigma_{\mathrm{F}_\mathcal{V}} \times \Sigma^{\mathrm{GEN}}) \cup (\Sigma_{\mathrm{F}_\mathcal{V}} \times \Sigma_{\mathrm{F}_\mathcal{V}}) \end{pmatrix} \cap \mathsf{po}^+$$

Then, we simply replace po in the SC axiom with vppo:

$$\mathit{acyclic}(\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr} \cup \mathsf{vppo}) \qquad (\overline{\mathrm{SC}})$$

We say that executions X satisfying $\overline{\mathrm{SC}}$ are Mixed-View SC, and write $\overline{\mathrm{SC}}(X)$.

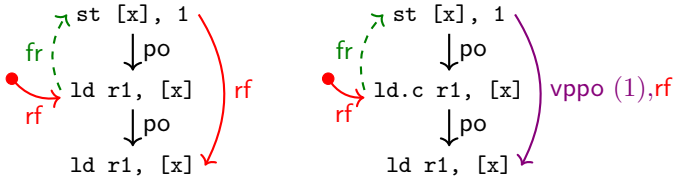
We say that executions X satisfying $\overline{\mathrm{SC}}$ are Mixed-View SC, and write $\overline{\mathrm{SC}}(X)$. The definition of $\overline{\mathrm{SC}}$ allows us to make a useful observation: because po has been replaced with vppo, it is straightforward to prove that in any execution X, where X has only events of generic view, that $\overline{\mathrm{SC}}(X) \iff \mathrm{SC}(X)$. When there is only the generic view $\mathsf{vppo} = \mathsf{po}$, so the $\overline{\mathrm{SC}}$ axiom can be re-written to be $\mathit{acyclic}(\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr} \cup \mathsf{po})$, which is the same as the original SC definition. For this reason, this change to the memory model can be considered an extension: all the existing single-view executions remain untouched by our change, but the model now incorporates weak behaviour that is possible when mixed view programming.

Note that there is no need to mimic the other changes needed to produce TSO. In our example simple architecture we have not considered optimisations like forwarding between different view caches, accordingly we have not weakened rf. Likewise, the view caches are non-coherent, and mixed-view pairs violate normal single-threaded semantics by construction, so there is no need for a modified SC-per-location axiom.

Finally, we add instruction fetches into the model in a straightforward way: we add an implicit load of instruction memory somewhere in program order between the execution of the associated instruction and the most recent instruction view fence preceding that instruction in program order. Every instruction fetch is modelled as being performed via the instruction view, and hence via the non-coherent instruction cache.

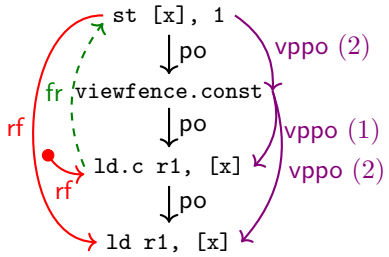
3.4.4 Example Executions

To illustrate the mixed-view SC model in action, we consider how some small litmus tests behave under both SC and $\overline{\text{SC}}$. We have labelled vppo edges with the rule that corresponds to them. We will take the set of views $\mathcal{V}$ to be $\{\text{GEN}, \text{CONST}\}$. We will use plain `ld` and `st` as load and store instructions of the GEN view respectively, and `ld.c` to be a CONST view load.



The sequence on the left shows three instructions: a store to `[x]`, followed by two consecutive loads of `[x]`. The fr and rf edges indicate that the loads return 0 (the implicit initial condition) and 1, respectively. This program has a cycle of fr;po, which is forbidden under the SC axiom. Indeed, standard single-threaded behaviour dictates that the middle load must see the value written by the store, as the second load does.

The sequence on the right shows a similar program, except now the middle load is a constant load (`ld.c`), and we use $\overline{\text{SC}}$ instead of SC. The same cycle is still present; however, this cycle is no longer forbidden under $\overline{\text{SC}}$! The store and middle load are of different views and not separated by a view fence, thus they are excluded from vppo. Architecturally, this scenario is exactly what could happen if the store passes through the generic cache while the load hits on stale data in the non-coherent constant cache, and hence it represents our mixed-view model working as intended. The second load does, however, need to observe the value written by the store, as it is made via the same view and hence is ordered after the store by vppo.



The behaviour of the second litmus test can be forbidden under $\overline{\text{SC}}$ by placing a constant view fence between the store and the middle load, as shown above. The constant view fence results in a pair of vppo edges being instantiated between the store and the middle load, and results in a $\mathsf{fr} \cup \mathsf{vppo}$ cycle that is forbidden under $\overline{\text{SC}}$. Indeed, as the constant view fence will invalidate stale data present in the constant cache, the subsequent constant load will fetch new data from generic memory; namely, the value written by the store. In the Mixed-View SC model the behaviour could be forbidden in one of two ways. A programmer may choose to put both writes in the same view, as in the program on the left below. This will ensure a coherence order edge, and give rise to the cycle in $\overline{\mathsf{fr}} \cup \mathsf{po}$ which is forbidden by the $\overline{\text{SC}}$ axiom. Alternatively, the programmer may choose to include a view fence. By including a view fence, the programmer has placed all the events in this program in vppo order. Note that the $\overline{\mathsf{co}}$ present in the fenced program is between accesses of different views. It must be present to satisfy the $\overline{\text{Coherence}}$ axiom, even though it creates more ordering than is needed for a per-location per-view total order.

3.5 Introduction to the PTX memory model

Before diving into the changes needed to extend PTX to support views, we first recap the baseline model. We recall the PTX memory model as formalised by NVIDIA Corporation (2019).

The PTX model, shown in Figure 28, ranges over *events*, *addresses*, and *scopes* in a candidate execution. These are represented by the sets Σ, $\mathcal{A}$, and $\mathcal{S}$, respectively. The event set Σ is decomposed into reads, writes, (memory) fences, and (execution) barriers, denoted by Σ_{Read}, Σ_{Write}, Σ_{Fence}, and Σ_{Barrier}, respectively. Each read and write has an associated address. Acquire events Σ_{Acq} and release events Σ_{Rel} form subsets of Σ_{Read} and Σ_{Write}, respectively. Σ_{Fence} is partitioned into sequentially-consistent, acquire, and release fences, denoted $\Sigma_{\text{F}_{SC}}$, $\Sigma_{\text{F}_{Acq}}$, and $\Sigma_{\text{F}_{Rel}}$ respectively. Each event has at most one program order successor (po), and exactly one scope.

The scope set $\mathcal{S}$ and the relation down form a tree of System, Devices, (Thread) Blocks, and Threads, denoted by $\mathcal{S}_{\text{System}}$, $\mathcal{S}_{\text{Device}}$, $\mathcal{S}_{\text{Block}}$, and $\mathcal{S}_{\text{Thread}}$, respectively. An example is depicted in

$$[X] \triangleq \{(a,a) \mid a \in X\} \qquad \text{(Set To Relation)}$$
$$\mathsf{r}^? \triangleq \mathsf{r} \cup \mathsf{id} \qquad \text{(Reflexive Closure)}$$
$$\mathit{scoped\_down}(X) \triangleq [X]; \mathsf{scope}; \mathsf{down}^*; \mathsf{start}; \mathsf{po}^*$$
$$\mathit{strong}(\mathsf{r}) \triangleq \mathsf{r} \cap \left\{ (a,b) \;\middle|\; \begin{pmatrix} a \in \mathit{scoped\_down}(b) \\ \wedge\, b \in \mathit{scoped\_down}(a) \end{pmatrix} \right\}$$

(a) Notation and functions

$$\mathsf{address} \subseteq (\Sigma \times \mathcal{A}) \qquad \mathsf{po} \subseteq (\Sigma \times \Sigma)$$
$$\mathsf{scope} \subseteq (\Sigma \times \mathcal{S}) \qquad \mathsf{down} \subseteq (\mathcal{S} \times \mathcal{S}) \qquad \mathsf{start} \subseteq (\mathcal{S}_{\mathsf{thread}} \times \Sigma)$$
$$\mathsf{dep} \subseteq (\Sigma_{\mathrm{Read}} \times \Sigma) \qquad \mathsf{sc} \subseteq (\Sigma_{\mathrm{F}_{SC}} \times \Sigma_{\mathrm{F}_{SC}})$$
$$\mathsf{co} \subseteq (\Sigma_{\mathrm{Write}} \times \Sigma_{\mathrm{Write}}) \qquad \mathsf{synchronizes} \subseteq (\Sigma_{\mathrm{Barrier}} \times \Sigma_{\mathrm{Barrier}})$$

(b) Primitive Relations (well-formedness constraints not shown)

$$\mathsf{same\_loc} \triangleq \mathsf{address}; \mathsf{address}^{-1}$$
$$\mathsf{po\_loc} \triangleq \mathsf{po} \cap \mathsf{same\_loc}$$
$$\mathsf{com} \triangleq \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co}^+$$
$$\mathsf{obs} \triangleq \mathit{strong}(\mathsf{rf} \cup \mathsf{rmw})$$
$$\mathsf{sync}_P \triangleq ([\Sigma_{\mathrm{F}_{Rel}}]; \mathsf{po}) \cup ([\Sigma_{\mathrm{Rel}}]; \mathsf{po\_loc})$$
$$\mathsf{sync}_Q \triangleq (\mathsf{po}; [\Sigma_{\mathrm{F}_{Acq}}]) \cup (\mathsf{po\_loc}; [\Sigma_{\mathrm{Acq}}])$$
$$\mathsf{sync} \triangleq [\Sigma_{\mathrm{Rel,F}_{Rel}}]; \mathit{strong}(\mathsf{sync}_P^?; \mathsf{obs}^+; \mathsf{sync}_Q^?); [\Sigma_{\mathrm{Acq,F}_{Acq}}]$$
$$\mathsf{cause}_{base} \triangleq \mathsf{po}^*; (\mathsf{sync} \cup \mathsf{synchronizes} \cup \mathsf{sc}); \mathsf{po}^*$$
$$\mathsf{cause} \triangleq \mathsf{obs}^*; (\mathsf{po\_loc} \cup {\mathsf{cause}_{base}}^*)$$

(c) Derived Relations

$$\mathit{acyclic}(\mathsf{rf} \cup \mathsf{dep}) \qquad \text{(No Thin Air)}$$
$$\mathit{acyclic}(\mathit{strong}(\mathsf{com}) \cup \mathsf{po\_loc}^+) \qquad \text{(Location SC)}$$
$$(\mathit{strong}(\mathsf{fr}); \mathit{strong}(\mathsf{co})) \cap \mathsf{rmw} = \emptyset \qquad \text{(Atomicity)}$$
$$\mathsf{same\_loc} \cap ([W]; \mathsf{cause}; [W]) \subseteq \mathsf{co}^+ \qquad \text{(Coherence)}$$
$$\mathit{irreflexive}(\mathsf{com}^?; \mathsf{cause}) \qquad \text{(Causality)}$$

(d) PTX(X) Axioms

Figure 28: The unextended Baseline PTX Memory Model (NVIDIA Corporation 2019)

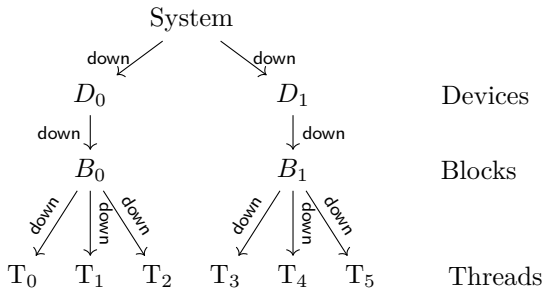


Figure 29: A sample scope hierarchy and down relation for threads T_0 to T_5

Figure 29. Each thread is related to exactly one event, the first event in that thread, by the start relation.

PTX incorporates scopes through the notion of *moral strength* (*strong*)[2], also known as mutual scope inclusion: two events are morally strong if they are each members of the set of events covered by the threads in the scope of the other. Synchronisation relations have effect only if the events in question are morally strong.

PTX also defines a set of auxiliary relations: *observation ordering* (obs), *synchronisation order* (sync), *synchronizes* (synchronizes), *SC fence order* (sc), and *causality order* (cause). The relations co, synchronizes, and sc are existentially quantified. co is a total order only over morally strong writes to the same address; this is notably weaker than the definition used in all standard CPU memory models, where co is scope-agnostic. synchronizes relates paired barriers, and sc is a total order over morally strong pairs of fences in $\Sigma_{\mathrm{F}_{SC}}$.

Synchronisation between threads is achieved via the cause relation, which uses morally strong release-acquire pairs to build chains of causality. The transitive use of cause- subcomponent cause_{base} provides *cumulativity* (i.e. transitivity).

Finally, the PTX memory model has five axioms. The *No Thin Air* axiom ensures that no values appear out-of-thin-air. The *Location SC* axiom ensures that all single-address program slices are well behaved within each individual scope. The *Atomicity* axiom provides atomicity for read-modify-write and reduction operations. The *Coherence* axiom ensures that properly-synchronised same-address stores are observed to occur in order. Finally, the *Causality* axiom describes how cause provides valid synchronisation.

[2]This *strong* relation is also known as scope inclusion (incl) in OpenCL. We use the name *strong* to match the NVIDIA terms of art.

Instruction	Description	Event set	View
`ld.wk.`s	Load	*Read*	GEN
`st.wk.`s	Store	*Write*	GEN
`ld.aq.`s	Acquire load	*Read*	GEN
`st.rl.`s	Release store	*Write*	GEN
`fence.sc.`s	SC fence	F_{SC}	GEN
`fence.aq_rl.`s	Acq/Rel fence	F_{AcqRel}	GEN
`ld.c`	Load constant	*Read*	CONST
`suld`	Load surface	*Read*	SURFACE
`sust`	Store surface	*Write*	SURFACE
`tld4`, `tex`	Texture load	*Read*	TEXTURE

Figure 30: Shorthand instructions for litmus tests. s parametrises instructions by scope.

3.5.1 PTX Instruction Interpretation

We model a straightforward interpretation of PTX instructions. Load instructions map into Read events, stores into Write events, and so on. Other details from the instruction encodings for GPU scopes, syntactic dependencies, and so on map into the model in the natural way. For convenience, we abstract away any details of PTX syntax unrelated to the memory model. A glossary of our PTX instruction shorthand is provided in Figure 30.

3.6 Views and Scopes in PTX

In our mixed-view PTX model, each view is associated with a particular scope: namely, the set of threads that can perform memory accesses through the specialised cache associated with that view. For example, if a view fence invalidates the texture cache for an SM, then all threads executing on the same SM will observe the effect of that invalidation, assuming they have properly synchronised with the thread that issued the view fence. Through the memory model concept of cumulativity (Alglave, Maranget and Tautschnig 2014), these causality links chain together to allow any threads to communicate via any views: standard scoped synchronisation is used to establish causality between threads, while view fences are used to establish causality between views. We formalise this as follows.

As before, we start by defining the set of views as $\mathcal{V}$, and the domain restriction Σ^v as the subset of events which have memory view v, and we add a new relation view relating each event to the view it uses when accessing memory. We introduce $\overline{\mathsf{same\_view}}$ as a relation between events of the same view: $\overline{\mathsf{same\_view}} \triangleq \mathsf{view}; \mathsf{view}^{-1}$.

In the architecture diagram motivating this model (Figure 24), each view cache lives within a particular SM, and is shared by all threads executing on that SM. The scope most closely

associated with the SM in hardware is the CTA/thread block, as all threads in a CTA are guaranteed to be scheduled on the same SM. We therefore constrain each non-generic view to be associated with a single thread block, and we denote this with the relation $\mathsf{view\_scope}$. This results in a view set constructed as a subset of all the possible architectural views a program may exercise:

$$\mathcal{V} \subseteq \{\textsc{Gen}\} \cup \bigcup_{s \in \mathcal{S}_{\mathsf{Block}}} \{\textsc{Const}_s, \textsc{Texture}_s, \textsc{Surface}_s\}$$

We constrain the view of each instruction either to be generic or to be one of the non-generic views associated with the thread block containing that instruction:

$$\mathsf{view\_scope} \subseteq \mathcal{V} \times \mathcal{S}$$
$$[\Sigma \setminus \Sigma^{\textsc{Gen}}]; \mathsf{view}; \mathsf{view\_scope} = \left(\mathsf{po}^{-1}\right)^*; \mathsf{start}^{-1}; \mathsf{down}^{-1}$$

This choice is tailored specifically to fit the GPU architecture shown in Figure 24; however, mixed-view models with other choices of $\mathsf{view\_scope}$ might be equally valid on underlying architectures with different view cache hierarchies.

3.6.1 Adjusting the PTX Auxiliary Relations

We then weaken the base PTX model to account for the weak behaviours observed in Section 3.2. First, to ensure eventuality of synchronisation accesses, we weaken "morally strong" to only relate events of the same view:

$$\overline{\mathit{strong}}(r) \triangleq r \cap \left\{ (a,b) \;\middle|\; \begin{pmatrix} a \in \mathit{scoped\_down}(b) \\ \wedge\, b \in \mathit{scoped\_down}(a) \\ \boxed{\wedge\, (a,b) \in \overline{\mathsf{same\_view}}} \end{pmatrix} \right\}$$

Second, we similarly weaken co to be an inherent total order over writes with matching scopes, addresses, *and* views:

$$\mathsf{stores}(a,s,v) \triangleq \begin{pmatrix} [s]; \left(\mathsf{down}^{-1}\right)^*; \mathsf{scope}^{-1} \\ \cap\, [s]; \mathsf{down}^*; \mathsf{scope} \\ \cap\, [a]; \mathsf{same\_loc}^{-1} \\ \cap\, \Sigma_{Write} \boxed{\cap([v]; \mathsf{view}^{-1})} \end{pmatrix}$$

$$\forall a \in \mathcal{A}, s \in \mathcal{S}, v \in \mathcal{V}. \mathit{total}(\overline{\mathsf{co}}, \mathsf{stores}(a,s,v))$$

Finally, following earlier discussion, we introduce one Block-scoped view fence for each non-generic view, and we introduce vppo in exactly the same way as Section 3.4.3:

$$\mathsf{vppo} \triangleq \begin{pmatrix} \overline{\mathsf{same\_view}} & \cup\ (\Sigma^{\mathrm{GEN}} \times \Sigma_{\mathrm{F}_\nu}) \\ \cup\ (\Sigma_{\mathrm{F}_\nu} \times \Sigma^{\mathrm{GEN}}) & \cup\ (\Sigma_{\mathrm{F}_\nu} \times \Sigma_{\mathrm{F}_\nu}) \end{pmatrix} \cap \mathsf{po}^{+}$$

We also define vppo_loc in the natural way:

$$\mathsf{vppo\_loc} \triangleq \mathsf{vppo} \cap \mathsf{same\_loc}$$

3.6.2 Adjusting Causality

Although many view caches are read-only, NVIDIA GPUs are able to perform atomic operations through the surface view. We therefore design our adjusted cause relation to support synchronisation being performed through any view, not just through the generic view. This adjustment turns out to be far more subtle than the others.

For example, we would like the outcome proposed in Figure 31 to be forbidden. Even though no view fences are present, the surface accesses will still be performed out of the same cache, and the release/acquire pairing will guarantee that the load accesses the texture cache after the store, so the synchronisation will still hold. We must therefore build the $\overline{\mathsf{cause}}$ relation such that it permits synchronisation through mixed views without losing track of which views have or have not been brought into alignment.

Likewise, we must take into account patterns such as the one in Figure 32 in which the view fence is not issued by the thread performing the non-generic memory access. Although the threads are distinct, the threads still share the same constant cache, and therefore the constant cache invalidation caused by the constant view fence in the first thread will ensure that the last thread indeed sees the value written by the first store. Therefore, the outcome in the figure must also be forbidden.

To account for the behaviours above, the $\overline{\mathsf{cause}_{base}}$ relation is formed as a chain of cause_{base} edges, constrained such that the *outermost* events must be performed via the same view of memory. In other words, the inner events (e.g., the acquire and release operations themselves) may be performed through other views of memory, but such synchronisation only has a direct effect on pairs of operations from the same view.

$$\overline{\mathsf{cause}_{base}} \triangleq \overline{\mathsf{same\_view}} \cap \left(\mathsf{po}^{*}; ((\mathsf{sync} \cup \mathsf{synchronizes} \cup \mathsf{sc}); \mathsf{po}^{*})^{+}\right)$$

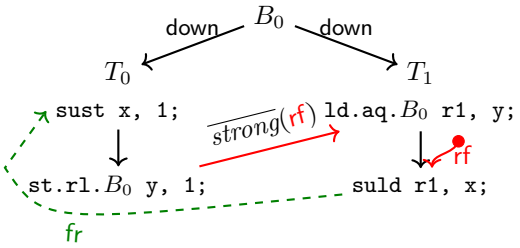


Figure 31: MP litmus test. This outcome is forbidden.

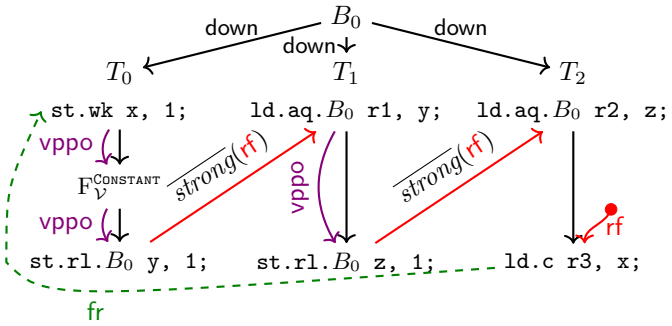


Figure 32: ISA2 litmus test. This outcome is forbidden.

The $\overline{\text{cause}}$ relation then transitively combines vppo and $\overline{\text{cause}_{base}}$, allowing the user to form cumulative chains of synchronisation even through multiple distinct views of memory. In particular, each individual synchronisation step ($\overline{\text{cause}_{base}}$) is performed within a single view, and then later these individual steps are combined via vppo to form a chain passing through as many views as are necessary. These carefully-chosen relations ensure that synchronisation can be performed correctly even for patterns such as Figure 32.

$$\overline{\text{cause}} \triangleq \text{obs}^*; (\text{vppo} \cup \overline{\text{cause}_{base}})^+$$

3.6.3 Adjusting the PTX Axioms

With vppo, moral strength, and cause properly adjusted, we can define the axioms of $\overline{\text{SC}}$ as shown in Figure 33. The Location SC and Atomicity axioms simply incorporate the notion of $\overline{strong}$ that accounts for views, and the Location SC axiom also only respects vppo_loc orderings, rather than all po_loc orderings. The Coherence relation states that only view-causally related stores to the same address are forced into coherence order. The Causality axiom states that no communication relation can contradict the view-aware notion of causality. Weak behaviours such as those described in Section 3.7.1 remain legal, however, as those examples do not use proper

$$
\begin{array}{ll}
\mathit{acyclic}(\mathsf{rf} \cup \mathsf{dep}) & (\overline{\text{No Thin Air}}) \\
\mathit{acyclic}(\overline{\mathit{strong}}(\mathsf{com}) \cup \mathsf{vppo\_loc}) & (\overline{\text{Location SC}}) \\
\overline{\mathit{strong}}(\mathsf{fr}); \overline{\mathit{strong}}(\overline{\mathsf{co}}) \cap \mathsf{rmw} = \emptyset & (\overline{\text{Atomicity}}) \\
\mathsf{same\_loc} \cap ([W]; \overline{\mathsf{cause}}; [W]) \subseteq \overline{\mathsf{co}}^{+} & (\overline{\text{Coherence}}) \\
\mathit{irreflexive}(\mathsf{com}^{?}; \overline{\mathsf{cause}}) & (\overline{\text{Causality}})
\end{array}
$$

Figure 33: Mixed-View PTX Axioms

view-aware synchronisation. Finally, given the unsolved nature of the problem it aims to prevent, we do not adjust the No Thin Air axiom.

3.6.4 Discussion: Forward Compatibility

We have tried to strike a balance between fitting the model to existing architecture documentation and empirical behaviour on one hand, and generality and forward compatibility on the other. For example, the model is not specialised for any one particular view. If the GPU were to add a new tightly-coupled accelerator with its own data-path and caches, then it would simply appear as a new view $v \in \mathcal{V}$. The instruction set may simply preclude, e.g., "constant stores" from ever appearing directly, but these restrictions do not need to appear in the memory model itself. We do currently assume that one view cache appears per multiprocessor, but any change to that assumption would simply require a corresponding modification to view_scope.

3.7 Validation

3.7.1 Litmus synthesis

We synthesise a set of "interesting" (which will be more precisely defined shortly) mixed-view litmus tests using an Alloy-based methodology derived in prior work by Lustig et al. (2017). Such tests in turn make excellent candidates for empirical testing of the model on forward-looking GPUs.

Interesting litmus tests are programs which have executions that are forbidden by the memory model, but such that applying some *weakening* function w to the execution changes the forbidden execution into a permitted execution (Lustig et al. 2017). A weakening function might, for example, demote a release write to a relaxed write – this corresponds to a program change from something like `st.rl x, 1` to `st.wk x, 1`.

	Causality		Coherence		Location SC		Atomicity	
	Plain	**Mixed**	**Plain**	**Mixed**	**Plain**	**Mixed**	**Plain**	**Mixed**
CoLB	6	21						
CoLB_W	2	5						
CoMP		13						
CoMP_R		6						
CoMP_W		3						
CoRR	2	7						
CoRW			2	7			2	2
CoRW+co					1	1		
CoS				10				
CoWR					2			
CoWW			1	4	1	1		
CoWW_R					1	1		
LB	6	21						
LB_R		3						
LB_W	2	5						
MP	6	21						
MP_W	2	5						
RW		3	1	1	1	1		
S			6	21				
WR	1	4			1	1		
WRC	6	12						
WWC			6	12				
Others	0	5	2	19	9	27	0	0
Total	33	134	18	74	16	34	2	2

Figure 34: Families of generated litmus tests. Plain tests contain only operations of the generic view, mixed tests contain operations on mixed views

The weakenings that we explore are:

- *Demote View* changes the view of some event to another view. This is the only new weakening.
- *Remove Event* removes an event.
- *Remove Dependency* removes a dependency edge.
- *Demote Release/Acquire* changes a release or acquire event into a write or read respectively.
- *Lower scope* changes some event's scope to a down-later scope, where possible.

Concretely, an interesting execution of the $\overline{\text{PTX}}$ model is defined to satisfy this formula:

$$interesting(X) \triangleq \neg\overline{\text{PTX}}(X) \wedge \forall w.\overline{\text{PTX}}(w(X))$$

We find interesting executions by exploring the axioms of the memory model one at a time. We generated all tests with up to five events, as this was the largest bound to finish within 48 hours. We omit $\overline{\text{No Thin Air}}$, the reason for this is twofold. First, this axiom did not change from the base model, and hence it is unaware of memory views. Consequently, view demotion never results in an execution forbidden by the $\overline{\text{No Thin Air}}$ axiom becoming legal. Secondly, with the ability to trivially form a large number of incoherent executions there are many Out Of Thin Air programs which are found using this method which are not actually very interesting at all. We also assume $\overline{\text{Coherence}}$ when generating for $\overline{\text{Causality}}$, as empirically this seemed to filter out a large number of tests built around obscure coherence synchronisation limitations rather than on programs synchronising in more causality-centric ways.

We categorise the interesting executions into multiple dimensions. In one dimension, we abstract away the views and categorise the tests using the standard taxonomy from the literature. In another, we characterise some tests according to PTX-specific variations of this taxonomy. Finally, we sort the tests into plain (generic only) and mixed-view buckets.

Hashing and abstraction. Programs synthesised with Alloy methodology are hashed into a human readable canonical representation. These canonicalised programs are stored in a set which grows as Alloy finds more interesting witnesses and new hashes are generated. Unique hashes are collected and printed. We take the unique hashes and abstract them down to a pattern of loads and stores over addresses in order to automatically bin them into coarse grained families of litmus tests. We report on the uniquely generated tests, which represent a fraction of the litmus test programs synthesised by Alloy.

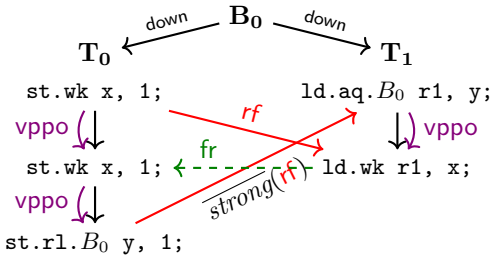
3.7.2 Synthesis results

Figure 34 tabulates the litmus tests synthesised by Alloy. We generated 313 litmus tests, representing all interesting programs with up to 5 events. 68% of these interesting tests are new tests in the suite utilising the mixed view extension. The first column of results, "Coherent Cause", corresponds to the $\overline{\text{Causality}}$ axiom generating tests with coherent executions as an assumption. Most of the tests synthesised in our mixed-view context appear to behave "as expected", i.e., they have the expected flavours of cause synchronisation and view mixing. This helps build confidence in the model.

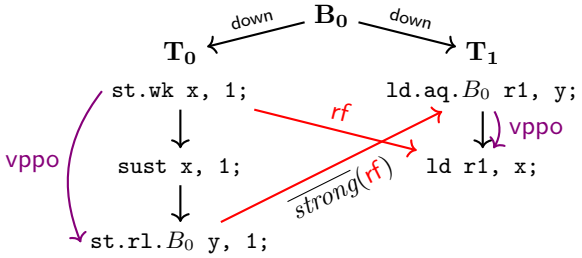
Litmus test synthesis was useful in catching a bug in an earlier version of our model, even when manual litmus test selection did not uncover it. Specifically, the test in question was a CoMP variant (see Figure 36): a one-address variant of the MP litmus test. Before we addressed all of the subtleties in replacing po_loc with vppo_loc, this execution was incorrectly forbidden in the mixed-view memory model. Although anecdotal, this example highlights the value of automation in memory model analysis and even in litmus test generation. The intuition for why it is an allowed execution is that the compiler and hardware have no reason to keep the surface and generic accesses to `x` in order. This led to our tweak of the $\overline{\text{Location-SC}}$ axiom to use vppo_loc instead of simply po_loc.

Common Program Variations. We observe that many tests are variations on common tests that already exist in the literature. These variations are a result of scope-based weak behaviours unique to the GPU and of the new mixed view model extension. In most cases, there is combinatorial expansion of both. For example, consider Figure 37. This scoped variant of CoWR is "interesting" in the GPU context, even though it seemingly contains CoWR as a smaller "interesting" subset. However, without the first load being part of the test, the two stores would no longer be morally strong, and hence they would not be forced into coherence order any longer. This highlights the GPU-specific aspect of litmus test generation.

Release sequences. As a PTX-specific subtlety, the prefix sync edges (sync_P) allow for a "release sequence" of a release followed by a write to the same location. This modification gives rise to a set of extended litmus tests in which a release operation is replaced with a release followed by a same-address write. The same holds true for acquire sequences. This is not directly related to views, but it does highlight an interesting quirk of the PTX ISA using both acquire/release operations and fences for synchronisation. We classify such tests via "_W" and "_R", respectively, in Figure 35. The remaining unclassified tests ("Others") have chains of



(a) Forbidden "interesting" execution. Any weakening of this program will yield an allowed execution.



(b) Allowed execution, where the program to the left has had a generic `st` event weakened to a constant `st.c` event.

Figure 35: A new coherence-variant of MP

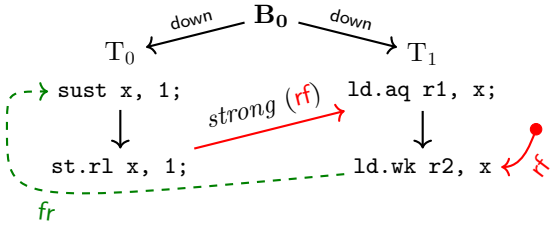


Figure 36: CoMP litmus test

coherence which interact with scopes in various permutations.

To study the impact of mixed views, let us consider MP. Our model generated 27 MP programs which are interesting with respect to the $\overline{\text{Causality}}$ axiom. Six of these are Generic-only single view programs, and one is single-view only. Two have no generic events, and the remaining programs have some mixture of generic and other views. Some of the programs are enumerated in Figure 38.

The release sequencing means we have a lot of common litmus tests which are extended with `st x _; st x _` patterns.

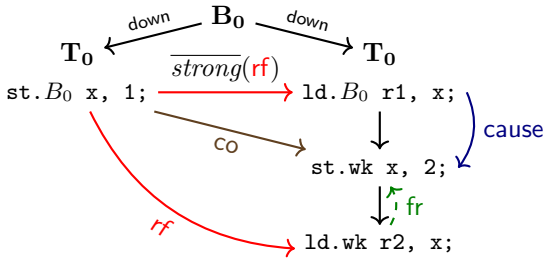


Figure 37: Due to GPU-specific model constraints, tests such as this one (a variant of CoWR) become "interesting". Specifically: the first load is needed in order to establish co, and hence fr, and hence the cycle that rules out this execution.

Coherence. The scopes and views of the memory model greatly increases the number of ways to generate incoherent observations of the same address. Any pair which is in same_loc, but not in $\overline{strong}$ may be observed to be incoherent.

We also split these tests by whether they exercise mixed view features of the model to demonstrate the way views increase the space of litmus tests due to these effects.

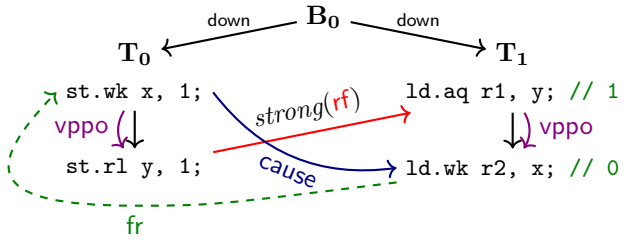
Special treatment of no thin air. As our measure of interesting execution is one where an execution must go from forbidden to permitted with *any* weakening of the program, we find that there are no programs which violate the no thin air axiom which can be weakened by views.

As the $\overline{\text{No Thin Air}}$ axiom definition does not involve views, simply dependencies and non-$\overline{strong}$ communication edges. As such, to generate executions which are interesting wrt. the no thin air axiom, we remove the restriction that they must become permitted executions with view-demotion.

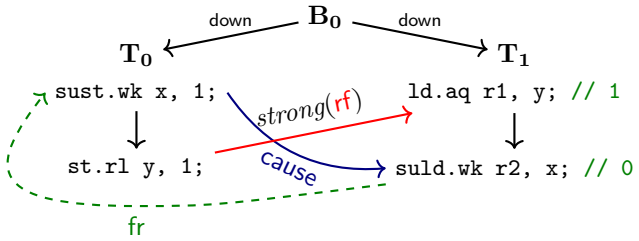
Limitations

As shown previously (Lustig et al. 2017), the generation of litmus tests using our methodology is super-exponential in time with the size of the test. Also, for this model in particular, the search space for tests is vast. In addition, a great deal of the tests were redundant with others already generated: Alloy found 32,037 interesting tests for the $\overline{\text{Causality}}$ axiom at bound 5, of which just 167 were unique. These limitations to some degree preclude generation of comprehensive test suites even of tests with six events. We were, however, able to generate complete suites for sizes up to 6 for specific known families (e.g., ISA2). We leave future work to find a more efficient way to generate larger "interesting" litmus tests automatically.

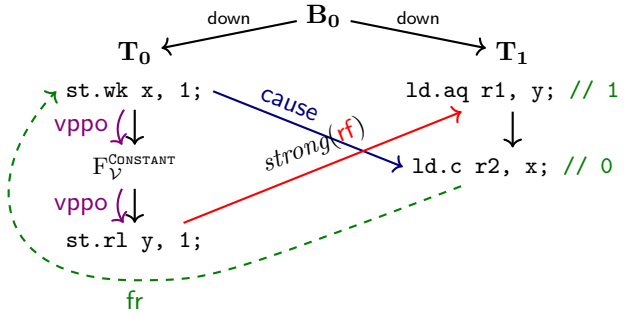
When generating tests which are interesting wrt. the $\overline{\text{Causality}}$ and $\overline{\text{Coherence}}$ axioms the solver would time out after 48 CPU hours. This is problematic with mixed view programming,



(a) Regular generic view MP



(b) Updating a surface and synchronising through generic memory



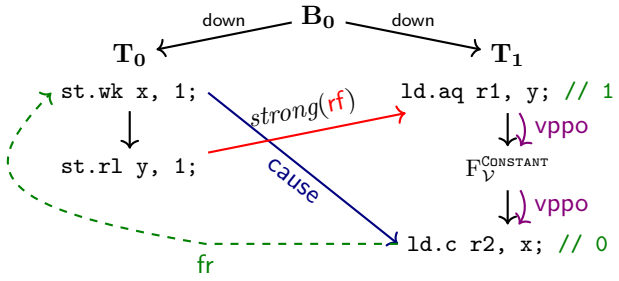
(c) Writing to a constant using generic operations, flushing the constant cache and synchronising through generic memory

Figure 38: Various interesting MP programs and their forbidden executions

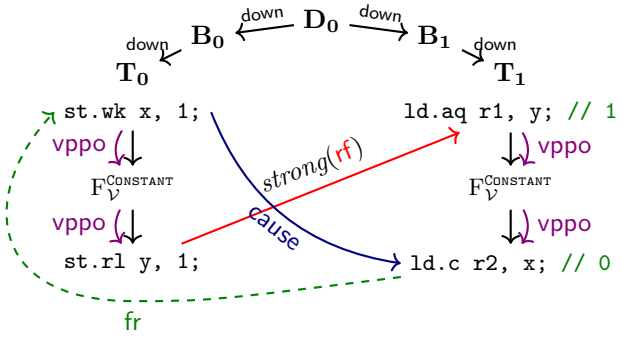
where view fences introduce additional events into otherwise short litmus tests.

To model ISA2 using 3 separate views, for example would require 12 events. 3 load/store pairs, and 6 view fences to synchronise the memory views. Synthesis of this test is unfortunately impractical with this methodology. This being said, direct observation of the test is possible with a short Alloy specification.

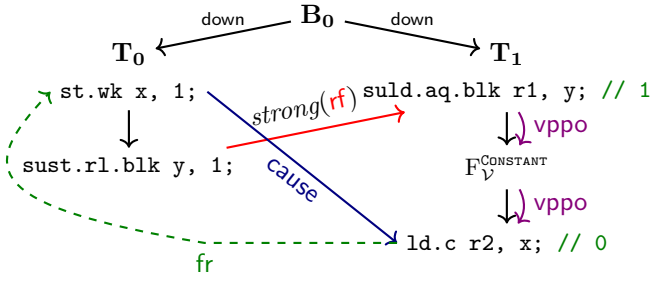
While this limitation is frustrating, the synthesis of litmus tests in this fashion is the state-of-the-art, and the method does cover a significant fraction of the tests in the literature.



(d) Writing to a constant using generic operations, synchronising through generic memory, and invalidating the constant cache



(e) Writing to a constant using generic operations, synchronising through generic memory, and invalidating the constant cache. This requires both view fences because of the views are in distinct blocks.



(f) A final hypothetical program: writing as generic, synchronising through surface, and loading as constant!

Figure 38: (continued) Various interesting MP programs and their forbidden executions

3.7.3 Model Meta-theory

In this section, we justify the validity of the memory model extension described in the previous section by empirically checking and then formally proving certain meta-theoretical properties about the model.

Methodology

We follow a methodology developed by previous work studying the NVIDIA PTX memory model (NVIDIA Corporation 2019). First, we encode the model and our theorems in Alloy (Jackson 2002), a relational model finding tool, using known axiomatic memory modelling techniques (Lustig et al. 2017; Wickerson et al. 2017). Second, we use Alloy to empirically test the theorems up to some reasonable bound (at least 5 events in all cases). This does not provide any guarantee of completeness, but it does provide a very useful sanity checking step that allows us to catch some false assertions in advance. Third, we use the `alloqc` tool (NVIDIA Corporation 2019) to automatically compile the model and assertions from Alloy into Coq (The Coq Development Team 2017), an interactive proof assistant. This compilation step elides the bounds used during model finding, allowing us to prove that the theorems hold for problems of any size. Finally, we fill in the (unbounded) proofs manually; proofs are verified with Coq.

A Sound and Complete Extension of PTX

Completeness. We show that all executions permitted by the original PTX memory model are also permitted in our extension of the memory model.

Theorem 3.7.4 (Completeness)**.** $\forall X.\mathrm{PTX}(X) \implies \overline{\mathrm{PTX}}(X)$

Proof. We show $\overline{PTX}$ one axiom at a time.
Location SC: $\overline{strong}(\mathsf{com}) \subseteq strong(\mathsf{com})$ trivially, and $\mathsf{vppo} \subseteq \mathsf{vppo\_loc}$, so a mixed Location SC cycle would imply a Location SC cycle, a contradiction.
Atomicity: follows from $\overline{strong}(\mathsf{com}) \subseteq strong(\mathsf{com})$.
No Thin Air: this is unchanged, and hence holds trivially.
Coherence: $\overline{\mathsf{cause}} \cap \mathsf{same\_loc} \subseteq \mathsf{cause}$; the non-obvious part is $\mathsf{vppo} \subseteq \mathsf{po\_loc}$, but this is covered by the Coherence axiom assumption of $\mathsf{same\_loc}$.
Causality: $\overline{\mathsf{cause}}$ is almost contained in cause:

$$\overline{\mathsf{cause}} \subseteq \mathsf{obs}^*; (\mathsf{po}^+ \cup \mathsf{cause}_{base}).$$

However, in the context of a mixed-view Causality cycle of $\mathsf{com}; \overline{\mathsf{cause}}$, we know that $(\mathsf{po}^+ \cup \mathsf{cause}_{base})$ must be contained in $\mathsf{same\_loc}$, and this gives us $\overline{\mathsf{cause}} \subseteq \mathsf{cause}$, thereby forming a plain Causality cycle, a contradiction. □

Soundness. We show that we do not introduce new executions to the existing memory model where there is exactly one view of memory being used. First we prove a lemma:

Lemma 1. $\mathcal{V} = \{\text{GEN}\} \implies \forall r. \overline{strong}(r) = strong(r)$

Proof. Everything but $\overline{\mathsf{same\_view}}$ is straightforward. $\overline{\mathsf{same\_view}}$ follows from the hypothesis that there is only one view. □

Now we prove soundness:

Theorem 3.7.5 (Soundness)**.** $\mathcal{V} = \{\text{GEN}\} \implies \forall X. \overline{\text{PTX}}(X) \implies \text{PTX}(X)$

Proof. We check for $\text{PTX}(X)$ one axiom at a time.
Location SC: $strong(\mathsf{com}) \subseteq \overline{strong}(\mathsf{com})$ by Lemma 1. $\mathsf{po\_loc}$ is in vppo since there is only one View. Therefore, every edge in $(strong(\mathsf{com}) \cup \mathsf{po\_loc})$ is in $(\overline{strong}(\mathsf{com}) \cup \mathsf{vppo})$.
Atomicity: follows from Lemma 1.
No Thin Air: this is unchanged, and hence holds trivially.
Coherence: We must show $\mathsf{cause} \subseteq \overline{\mathsf{cause}}$.

- $strong(\mathsf{obs}) \subseteq \overline{strong}(\mathsf{obs})$ due to Lemma 1
- $\mathsf{po\_loc} \subseteq \mathsf{vppo}$, as before
- $\mathsf{cause}_{base} \subseteq \overline{\mathsf{cause}_{base}}$ also follows from Lemma 1

Causality: $\mathsf{cause} \subseteq \overline{\mathsf{cause}}$; the rest is straightforward. □

SC-for-DRF

SC-for-DRF theorems form the basis of most software-level programming models, since software models generally make clear distinctions between synchronisation operations and normal (non-synchronisation) operations. In contrast to most software-targeted models, and even in contrast to the HSA memory model (Foundation 2015), the PTX memory model does not give so-called "catch-fire" semantics for racy programs: all well-formed programs have well-defined memory model behaviour under PTX. In hardware models, even synchronisation operations have often been decomposed into finer-grained operations that are themselves racy.

For example, if load-acquire and store-release operations are decomposed into plain loads, plain stores, and fences on a given hardware ISA, then the load-acquire/store-release may not be racy at a software level, but the plain load and plain store may still be racy at a hardware level. Prior work has proposed some model-/algorithm-specific notions of architecture-level data race freedom, such as the use of Triangular Race Freedom to study the behaviour of locks under x86-TSO (Sewell et al. 2010). However, this technique does not generalise in an obvious way to other hardware models or to more generic software situations.

Similarly, from an ISA memory model perspective, failed attempts to acquire a lock that is held by another thread will often be considered to race with the eventual unlock store. Such behaviour is generally accounted for by asserting that such "racy" failed lock acquisition will cause no other side effects. For example, the lock acquisition algorithm may simply spin until the lock is successfully acquired. As long as the loads for the failed acquisitions have no other side effects, they will not adversely affect the memory behaviour. Executions such as these, with successful acquisitions only, are the behaviours we study with our SC-for-DRF theorem.

Nevertheless, as a weak memory model, PTX does not in general produce only sequentially consistent outcomes when running programs with data races. In order to ensure the generality of our proposed extension, we prove a basic sequential-consistency-for-data-race-free (SC-for-DRF) property for mixed-view PTX.

To start the proof, we define SC as in Section 3.4, but with com replacing its components:

$$\mathsf{SC} \triangleq \mathit{acyclic}(\mathsf{com} \cup \mathsf{po\_loc})$$

We also define a mixed-view notion of happens-before:

$$\overline{\mathsf{hb}} \triangleq (\mathsf{vppo} \cup \overline{\mathsf{cause}_{base}})^{+}$$

We then define executions to be mixed-racy if there exists a pair of memory accesses that are to the same address, such that at least one is a write, and that are not separated by $\overline{\mathsf{hb}}$:

$$\begin{aligned}\overline{\mathsf{DRF}} \triangleq \forall a \in \mathit{Write}, b \in (\mathit{Write} \cup \mathit{Read}).\\ (a \neq b) \wedge (a, b) \in \mathsf{same\_loc}\\ \implies (a, b) \in \overline{\mathsf{hb}} \vee (b, a) \in \overline{\mathsf{hb}}\end{aligned}$$

Notably, by this definition, it is possible to have a data race even between memory accesses from a single thread but performed via different views. This reflects the fact that the common notion

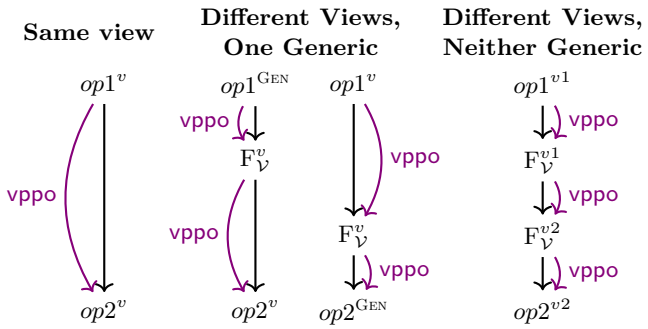


Figure 39: Avoiding intra-thread data races

of single-threaded behaviour need not be respected when mixing views.

Our SC-for-DRF result shows that programmers can avoid data races and restore sequentially consistent behaviour by doing two things. First, they must ensure that all intra-thread pairs of memory accesses to the same address are separated by a sufficient number of view fences, per Figure 39. Second, they must use proper cause_{base} (i.e., acquire-release) synchronisation between threads. This combination ensures that only sequentially consistent outcomes will be observed.

Theorem 3.7.6 (SC-for-DRF). $\forall X.\overline{\text{PTX}}(X) \wedge \overline{\mathsf{DRF}} \implies \mathsf{SC}$

Proof. First, we define an intermediate relation:

$$\mathsf{cause}_{mid} \triangleq \mathsf{po}^{+} \cup (\mathsf{po}^{*}; (\mathsf{synchronizes} \cup \mathsf{sc} \cup \mathsf{sync}); \mathsf{po}^{*})^{+}$$

Second, we show by induction that

$$\mathsf{DRF} \implies (\overline{\mathit{strong}}(\mathsf{com}) \cup \mathsf{po\_loc}) \subseteq \mathsf{cause}_{mid}$$

For the base case, by the DRF assumption, if $(a, b) \in \mathsf{com}$, then either $(a, b) \in \overline{\mathsf{hb}}$ or $(b, a) \in \overline{\mathsf{hb}}$, but the latter contradicts the Causality axiom. $\overline{\mathsf{hb}} \subseteq \mathsf{cause}_{mid}$ and $\mathsf{po\_loc} \subseteq \mathsf{po}^{+}$ are straightforward. The inductive step follows naturally as well due to transitivity.

Third, as a consequence of the second step, a Location SC-violating $(\overline{\mathit{strong}}(\mathsf{com}) \cup \mathsf{po\_loc})$ cycle implies a cycle of cause_{mid}, which can be decomposed in one of two ways. First, a po^{+} cycle would be malformed. Second, $\mathsf{cause}_{mid} \subseteq \mathsf{cause}_{base}$ since the start and end are the same event and hence trivially in same_view, and then a cause_{base} cycle would contradict the Causality axiom. □

3.8 Conclusion

Use cases for mixed-view GPU programming are fast emerging. Low-latency kernel launch, compute/graphics interoperability, and hardware accelerator integration all depend on the ability to safely synchronise on and communicate through memory, even when that memory is accessed via specialised compute engines and cache hierarchies. Unfortunately, the GPU memory models today do not provide well-defined semantics for such programs, leaving gaps in the programming model.

We propose an elegant approach to extending memory models to provide well-defined semantics for mixed-view programming. To our knowledge, our approach provides the first general-purpose, compute-oriented GPU memory model to directly address non-generic views of memory. Our model takes empirical observations and official documentation of expected weak memory behaviour into account. It also proposes "view fences", readily-implementable mechanisms for synchronising non-generic memory views, and incorporates them into the model as a means of restoring sanity. We perform empirical analysis of interesting and novel families of litmus tests generated specifically to stress this model, and we also prove an SC-for-DRF theorem that allows programmers to write programs with confidence. As such, our proposed memory model successfully fills the GPU mixed-view programming model gap. This will open the door for a range of new software interoperability paradigms and hardware acceleration mechanisms for future GPU generations to come.

Chapter 4

Mechanising a Thin-Air Solution

This chapter presents PrideMM, the first tool for automatically evaluating a thin-air free memory model over a corpus of litmus tests.

In recent years a new class of memory model has emerged. These models capture the complex relationship between the aggressive optimisations of programming languages like C++ and Java while not permitting thin-air executions. Thin-air free models previously required hand proof and manual pen-and-paper evaluation to determine what outcomes would be permitted for a given litmus test. PrideMM is the first tool for automatically evaluating a thin-air free model, allowing quick bug discovery and mechanised validation of the model against real-world examples.

We use new solver technology to develop a tool for evaluating the thin-air free memory model of Jeffrey and Riely (2016). Jeffrey and Riely (2016) presented a "game semantics" model for exploring executions of aggressively optimised programming languages like Java and C++. In this semantics the "player" attempts to make progress towards a desired execution, while the "opponent" attempts to thwart their efforts. The exact details of this mechanism will be drawn out later. The end result is an interaction between player and opponent where the player must find some move which will make progress towards their goal for any move that the opponent makes.

Axiomatic memory models work by applying a set of constraints to individual executions and giving a determination as to whether that execution is permitted (Alglave, Maranget and Tautschnig 2014; Batty 2011; Lamport 1979). Conversely each of the existing thin-air free models have some mechanism for exploring many possible execution states (Kang et al. 2017; Pichon-Pharabod and Sewell 2016; Jeffrey and Riely 2016; Chakraborty and Vafeiadis 2019). A key insight about thin-air free models then, is that they use higher-order reasoning. The

J+R model is no exception: the interaction between player and opponent creates a higher-order exploration of program states than simple single-execution axiomatic models utilise.

As some background on memory model tools, it is useful to consider the two common approaches. One relies on ad hoc algorithms to evaluate a litmus test under a particular memory model (Alglave, Maranget and Tautschnig 2014; Batty et al. 2011; Gray et al. 2015; Paviotti et al. 2020), and the other encodes the mathematics of the memory model into a satisfiability (SAT) problem (Lustig et al. 2017; Wickerson et al. 2017; Torlak, Vaziri and Dolby 2010).

Ad hoc. Ad hoc algorithms constrain the expression of a memory model, and substantial changes to a model require re-development of the algorithm. As discussed in §2.2.1, Herd (Alglave, Maranget and Tautschnig 2014) implements an algorithm for axiomatic memory models, and provides a common framework for expressing these models by making assumptions about relations like program order and coherence order. Herd's algorithm is designed to enumerate the possible executions of a program, and independently determine whether each execution satisfies the rules of the axiomatic memory model. As this check is done on a per-execution basis, it suffers the same problem that axiomatic memory models as a whole do, individual executions do not contain enough information to reconstruct which dependencies are real dependencies and which are false dependencies, as discussed in §2.3. So, while thin-air free models like RC11 or hardware models like ARMv8 can be implemented in Herd, thin-air free models which allow for aggressive program optimisation – as is common in C++ and Java – do not appear to be compatible.

SAT. Using automatic solving based on the mathematical expression of the model is more flexible than ad-hoc algorithms. Thin-air free models' state exploration appears to be higher-order, with all the solutions to the thin-air problem doing some flavour of forall-exists exploration of the program behaviour. The higher order models of thin-air free solutions can even be encoded into SAT – as higher order logic can be reduced to SAT over finite models. Unfortunately, reduction from higher order logic to SAT involves exponential blow up, yielding a formulae which are too large to solve with existing solver technology.

QBF. PrideMM provides a DSL for expressing memory models in Second Order (SO) Logic. SO logic allows the expression of more complex formulae than those required to capture axiomatic memory models: in particular SO logic allows alternation of quantification. Where axiomatic models only existentially quantify relations like coherence order for a single execution, thin-air free models must quantify over all executions, and then quantify for each execution the existence of relations like coherence. SO logic is in a sweet spot of expressiveness and simplicity for building

automated memory model tools. It is expressive enough to encode complex memory models which can quantify over a range of executions, yet simple enough allow the use of automated solving techniques. PrideMM therefore allows us to answer questions about new thin-air free memory models quickly, by allowing many litmus tests to be run in what would otherwise be a time consuming manual process. PrideMM enables a modify-execute-evaluate pattern of memory model development, where changes can be quickly implemented and tested.

SO logical formulae which use alternating quantification are notated with the quantification ahead of "SO", for example formulae that only use existential quantification are said to be in the ∃SO fragment. This is a useful shorthand for expressing the complexity of a formula. Axiomatic memory models are definable in ∃SO in a natural way and one can evaluate them using SAT solvers. We demonstrate this facility of PrideMM for a realistic C++ memory specification (Lahav et al. 2017), reproducing previous results (Wickerson et al. 2017; Torlak, Vaziri and Dolby 2010). As we have discussed, unlike axiomatic models some memory models are naturally formulated in higher-order logic. Concretely, the memory model of Jeffrey and Riely (2016) (J+R) comes with a formalisation, in the proof assistant Agda (Bove, Dybjer and Norell 2009), that clearly uses higher-order features. We observed that the problem of checking whether a program execution is allowed by J+R can be reduced to the model checking problem for SO. In general, going from higher-order to second-order involves an exponential blow-up, but in the case of the Jeffrey-Riely model the reformulation can be done with only polynomial size change.

To illustrate the need for SO logic in building an automatic tool for the J+R model, the formula at the core of the model is depicted below. The exact details will be explained in more depth later, here it is sufficient to note the pattern of quantification, highlighted below.

$$\mathsf{JR}_n \coloneqq \underline{\exists X}\left(\mathsf{TC}_n(\underline{\mathsf{AeJ}_n})(\emptyset, X) \wedge \mathsf{F}(X)\right)$$

$$\mathsf{AeJ}_n(P,Q) \coloneqq \begin{cases} \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \wedge \\ \underline{\forall X}\left(\mathsf{TC}_n(\mathsf{AJ})(P,X) \rightarrow \underline{\exists Y}\left(\mathsf{TC}_n(\mathsf{AJ})(X,Y) \wedge \mathsf{J}(Y,Q)\right)\right) \end{cases}$$

Observe that the formula JR_n is in ∃∀∃SO—there is alternation of the ∃ and ∀ quantification in the structure of the functions. In practice, this means that it is not possible to use SAT solvers, as the nested quantifiers would have to be expanded into a single level of ∃ or ∀, and that would involve an exponential explosion. That motivates our development of an SO model checker. It is known that SO captures the polynomial hierarchy (Libkin 2004, Corollary 9.9), and the canonical problem for the polynomial hierarchy is quantified satisfiability. Hence, we built our SO model checker on top of a quantified satisfiability solver (QBF solver), QFUN (Janota 2018).

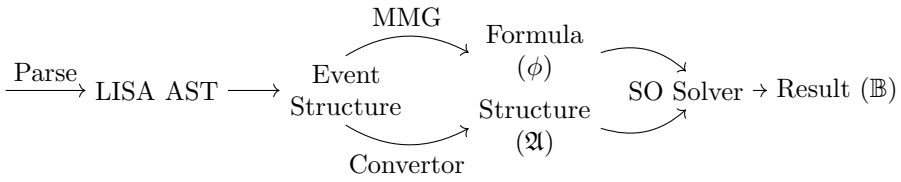


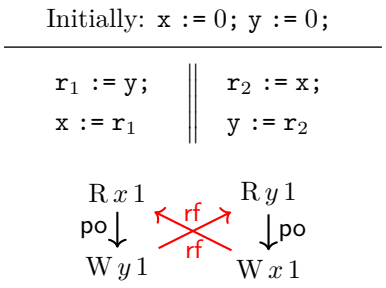
Figure 40: From a LISA test case to a Y/N answer, given by the SO solver.

4.1 Architecture of PrideMM

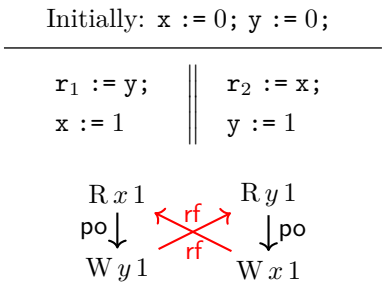
Figure 40 shows the architecture of PrideMM. The input is a litmus test written in the LISA language, and the output is a boolean result. LISA is a programming language that was designed for studying memory models (Alglave and Cousot 2016). We use LISA for its compatibility with the state-of-the-art memory model checker Herd7 (Alglave, Maranget and Tautschnig 2014), allowing us to later contrast PrideMM with Herd (§4.5). We transform the input program into an event structure (Winskel 1987), which is compatible with the J+R memory model, and contains all the information necessary for evaluating axiomatic models. The memory model generator (MMG) is an interchangeable component which produces the SO formula that encodes a memory model. For some memory models (§4.4.1, §4.4.2, §4.4.3) that Herd7 can handle as well, the formula is fixed and does not depend on the input program. For the J+R MMG (§4.4.4) the SO formula changes dependent on the input program, in this case the change relates to the size of the program. Finally, both the second-order structure and the second-order formula are fed into a solver, giving a verdict for the litmus test.

4.1.1 Developing a memory model in SO logic

Memory models describe the executions allowed by a shared-memory concurrent system; for example, under *sequential consistency* (SC) (Lamport 1979) memory accesses from all threads are interleaved and reads take their value from the most recent write of the same variable. Processor speculation, memory-subsystem reordering and compiler optimisations lead mainstream languages and processors to violate SC, and we say such systems exhibit *relaxed concurrency*. Relaxed concurrency is commonly described in an *axiomatic* specification (e.g. SC, ARM, Power, x86, C++ specifications (Lamport 1979; Deacon and Alglave 2019; Maranget, Sarkar and Sewell 2012; Sewell et al. 2010; Batty et al. 2011; Alglave, Maranget and Tautschnig 2014)), where each program execution is represented as a graph with memory accesses as vertices, and edges representing program structure and dynamic memory behaviour. A set of axioms permit some execution graphs and forbid others.



(a) LB+datas: observing $\mathtt{r}_1 = \mathtt{r}_2 = 1$ is a thin-air execution of this program, and must be forbidden.



(b) LB: observing $\mathtt{r}_1 = \mathtt{r}_2 = 1$ must be allowed. The 1/1 outcome is observable on some hardware, and can also be observed as a result of common compiler optimisations.

Figure 41: The 1/1 executions of LB and LB+datas are indistinguishable in the axiomatic setting.

Recall from §2.3 the LB litmus tests. When boiled down to executions it was impossible to tell the difference between a forbidden thin-air execution of LB+datas and an allowed weak execution of LB+false-dep. The programs are recalled in Figure 41, with a $\mathtt{r}_1 = \mathtt{r}_2 = 1$ outcome execution drawn for each.

Event structures capture the necessary information. A new class of specifications aims to fix this by ordering only real dependencies (Kang et al. 2017; Jeffrey and Riely 2016; Pichon-Pharabod and Sewell 2016; Chakraborty and Vafeiadis 2019). With a notable exception (Kang

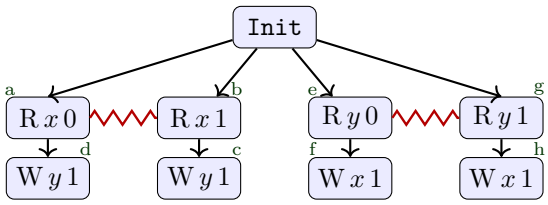


Figure 42: Event structure representation of LB.

et al. 2017), these specifications are based on *event structures*, where all paths of control flow are represented in a single graph. Figure 42 presents the event structure for LB. Program order is represented by arrows ($\rightarrow$). Conflict ($\rightsquigarrow$) links events where only one can occur in an execution (the same holds for their program-order successors). For example, on the left-hand thread, the load of `x` can result in a read of value 0 (event a) or a read of value 1 (event b), but not both. Conversely, two subgraphs unrelated by program-order or conflict, i.e. $\{a, b, c, d\}$ and $\{e, f, g, h\}$, represent two threads in parallel execution.

The event structure in Figure 42 also shows that regardless of the value read from `x` in the left-hand thread ($\{a, b, c, d\}$), there is a write to `y` of value 1. The apparent dependency from the load of `y` is false and could be optimised away. The similar argument follows for `y`/`x` in the right-hand thread ($\{e, f, g, h\}$). Memory models built above event structures can recognise this pattern and permit relaxed execution.

The Jeffrey and Riely model. As a recap from §2.3.2, the Jeffrey and Riely model (J+R) is built above event structures and correctly identifies false dependencies (Jeffrey and Riely 2016). Conceptually, the model is related to the Java memory model (Manson, Pugh and Adve 2005): in both, one constructs an execution stepwise, adding only memory events that can be *justified* from the previous steps. The sequence captures a causal order that prevents cycles with real dependencies. While Java is too strong, J+R allows writes that have false dependencies on a read to be justified before that read. To do this, the model recognises confluence in the program structure: regardless of the execution path, the write will always be made. As we saw in the introduction, J+R involves an alternation of quantification that current ad hoc and SAT-based tools cannot efficiently simulate. This alternation of quantification is core to how the J+R model explores multiple executions. Fortunately, while SAT tools cannot efficiently evaluate the J+R model, the problem is amenable to QBF solvers.

4.1.2 Developing SC in SO Logic

To introduce the SO language used to build memory models in PrideMM it is useful to first encode a simple memory model. The SC memory model can be expressed as an axiomatic model (Alglave, Maranget and Tautschnig 2014) using *coherence order*, a per-variable total order of write events. An execution is allowed if there exists a reads-from relation rf and a coherence order co such that the transitive closure of $\mathsf{rf} \cup \mathsf{co} \cup (\mathsf{rf}^{-1}; \mathsf{co}) \cup \mathsf{po}$ is acyclic. Here, po is the (fixed) program-order relation, and it is understood that co and rf satisfy certain further axioms.

In our setting, we describe the sequentially consistent specification as follows. We represent

rf and co by existentially-quantified SO arity-2 variables Y_{rf} and Y_{co}, respectively. For example, to say $(x, y) \in \mathsf{co}$, we use the formula $Y_{\mathsf{co}}(x, y)$. The program order po is represented by an interpreted arity-2 symbol $<$.

Then, the SO formula that represents $\mathsf{rf} \cup \mathsf{co} \cup (\mathsf{rf}^{-1}; \mathsf{co}) \cup \mathsf{po}$ is

$$\mathsf{R}(y, z) \;\coloneqq\; Y_{\mathsf{rf}}(y, z) \vee Y_{\mathsf{co}}(y, z) \vee \exists x \left(Y_{\mathsf{rf}}(x, z) \wedge Y_{\mathsf{co}}(x, y) \right) \vee (y < z)$$

Disjunction between each predicate is equivalent to a union in the relational algebra notation normally used in axiomatic memory models.

The definition from above should be interpreted as a macro expansion rule: the left-hand side $\mathsf{R}(y, z)$ is a combinator that expands to the formula on right-hand side. To require that the transitive closure of R is acyclic we require that there exists a relation that includes R ($\mathsf{sub}^2(\mathsf{R}, Z)$), is transitive (trans), and irreflexive (irrefl):

$$\exists Z \left(\mathsf{sub}^2(\mathsf{R}, Z) \wedge \mathsf{trans}(Z) \wedge \mathsf{irrefl}(Z) \right)$$

The precise definitions of these combinators will be introduced in the next section. For now note that common constructions in axiomatic memory models have natural equivalents in SO logic. Each piece of relational algebra can be easily written as a second-order term.

To represent programs and their behaviours uniformly for all memory specifications in section 4.4, we use event structures. Event structures can be seen as an overlay of all possible executions, so they encode all the information necessary to check executions in the axiomatic setting, and they are the basis for the J+R memory model.

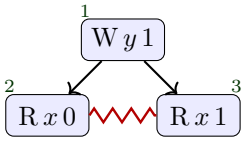
We have been using the symbol $\leq$ for program order so far, but have not precisely specified where this is encoded. Our second order model checking requires two parts, a logical formula (ϕ) which encodes the memory model and a logical structure ($\mathfrak{A}$) which contains fixed relations such as program order. In PrideMM we translate an event structure into a handful of relations:

- Events (arity 1) contains a list of each event in the memory model. This will be the "universe" of members that quantification can range over, and is notated as A.
- Program order (arity 2) relates events which follow in program order. We have been using $x \leq y$ for this symbol so far, but where it is useful to disambiguate we can also use $\texttt{ord}(x, y)$.
- Conflict (arity 2) relates events which are in conflict in the event structure.
- Writes (arity 1) is a set of all the events which are writes.

- Reads (arity 1) similarly is a set of all the events which are reads.
- Same value (arity 2) relates events with the same value in their label.
- Same location (arity 2) relates events with the same location in their label.

A few more relations are required in the details of the J+R model, but those will be introduced later in Section 4.4.4.

Consider then, the following event structure, and its corresponding structure $\mathfrak{A}$.



This yields a structure $\mathfrak{A}$ as follows:

$$
\begin{aligned}
A &\coloneqq \{1,2,3\} \\
\texttt{ord} &\coloneqq \{(1,2),(1,3)\} \\
\texttt{cnf} &\coloneqq \{(2,3),(3,2)\} \\
\texttt{write} &\coloneqq \{1\} \\
\texttt{read} &\coloneqq \{2,3\} \\
\texttt{same\_value} &\coloneqq \{(1,3)(3,1)\} \\
\texttt{same\_location} &\coloneqq \{(2,3)(3,2)\}
\end{aligned}
$$

Once we have the program and its behaviour represented as a logic structure $\mathfrak{A}$ and the memory model represented as a logic formula ϕ, we ask whether the structure satisfies the formula, written $\mathfrak{A} \models \phi$. In other words, we have to solve a model-checking problem for second-order logic, which reduces to QBF solving because the structure $\mathfrak{A}$ is finite.

4.2 Preliminaries

To introduce the necessary notation, we recall some standard definitions (Libkin 2004). A (finite, relational) *vocabulary* σ is a finite collection of *constant symbols* ($\mathtt{1}, \ldots, \mathtt{n}$) together with a finite collection of *relation symbols* ($\mathtt{q}, \mathtt{r}, \ldots$). A (finite, relational) *structure* $\mathfrak{A}$ *over vocabulary* σ is a tuple $\langle A, Q, R, \ldots \rangle$ where $A = \{1, \ldots, n\}$ is a finite set called *universe* with several distinguished relations $Q, R, \ldots$ We assume a countable set of *first-order variables* (x, y, …), and a countable

set of *second-order variables* (X, Y, ...). A *variable* α is a first-order variable or a second-order variable; a *term* t is a first-order variable or a constant symbol; a *predicate* P is a second-order variable or a relation symbol. A (second-order) *formula* ϕ is defined inductively: (a) if P is a predicate and $t_1, \ldots, t_k$ are terms, then $P(t_1, \ldots, t_k)$ is a formula[1]; (b) if ϕ_1 and ϕ_2 are formulae, then $\phi_1 \circ \phi_2$ is a formula, where $\circ$ is a boolean connective; and (c) if α is a variable and ϕ is a formula, then $\exists \alpha\, \phi$ and $\forall \alpha\, \phi$ are formulae. We assume the standard satisfaction relation $\models$ between structures and formulae.

The logic defined so far is known as relational SO. If we require that all quantifiers over second-order variables are existential, then we obtain a fragment known as $\exists$SO. For example, the SC specification of section 4.1.2 is in $\exists$SO.

The Model Checking Problem. Given a structure $\mathfrak{A}$ and a formula ϕ, determine if $\mathfrak{A} \models \phi$. We assume that the relations of $\mathfrak{A}$ are given by explicitly listing their elements. The formula ϕ uses the syntax defined above.

Combinators. We will build formulae using the combinators defined below. This simplifies the presentation, and directly corresponds to an API for building formulae within PrideMM.

$$
\begin{aligned}
\mathsf{sub}^k(P^k, Q^k) &\coloneqq \forall \vec{x}\, \big(P^k(\vec{x}) \to Q^k(\vec{x})\big) & \mathsf{id}(x, y) &\coloneqq (x = y) \\
\mathsf{eq}^k(P^k, Q^k) &\coloneqq \forall \vec{x}\, \big(P^k(\vec{x}) \leftrightarrow Q^k(\vec{x})\big) & \mathsf{inv}(P^2)(x, y) &\coloneqq P^2(y, x) \\
\mathsf{seq}(P^2, Q^2)(x, z) &\coloneqq \exists y\, \big(P^2(x, y) \wedge Q^2(y, z)\big) & \mathsf{irrefl}(P^2) &\coloneqq \forall x\, \neg P^2(x, x) \\
\mathsf{inj}(P^2) &\coloneqq \mathsf{sub}^2\big(\mathsf{seq}(P^2, \mathsf{inv}(P^2)), \mathsf{id}\big) & \mathsf{or}(\mathsf{R}, \mathsf{S})(x, y) &\coloneqq \mathsf{R}(x, y) \vee \mathsf{S}(x, y) \\
\mathsf{trans}(P^2) &\coloneqq \mathsf{sub}^2\big(\mathsf{seq}(P^2, P^2), P^2\big) & \mathsf{maybe}(\mathsf{R})(x, y) &\coloneqq \mathsf{or}(\mathsf{id}, \mathsf{R})(x, y)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{acyclic}(P^2) &\coloneqq \exists X^2\, \big(\mathsf{sub}^2(P^2, X^2) \wedge \mathsf{trans}(X^2) \wedge \mathsf{irrefl}(X^2)\big) \\
\mathsf{TC}_0(\mathsf{R}) &\coloneqq \mathsf{eq}^1 \\
\mathsf{TC}_{n+1}(\mathsf{R})(P^1, Q^1) &\coloneqq \mathsf{eq}^1(P^1, Q^1) \vee \exists X^1\, \big(\mathsf{R}(P^1, X^1) \wedge \mathsf{TC}_n(\mathsf{R})(X^1, Q^1)\big)
\end{aligned}
$$

By convention, all quantifiers that occur on the right-hand side of the definitions above are over fresh variables. Above, P^k and Q^k are arity-k predicates, x and y are first-order variables, $\vec{x}$ is a sequence of first order variables of appropriate length for the arity of the second order variables to which they are applied, and R and S are combinators.

Let us discuss two of the more interesting combinators: acyclic and TC. A relation P is acyclic if it is included in a relation that is transitive and irreflexive. We remark that the definition of acyclic is carefully chosen: even slight variations can have a strong influence on the runtime of solvers (Janota, Grigore and Manquinho 2017). The combinator TC for bounded transitive

[1] we make the usual assumptions about arity

closure is interesting for another reason: it is higher-order — applying an argument (R) relation in each step of its expansion. By way of example, let us illustrate its application to the subset combinator sub^1.

$$
\begin{aligned}
&\mathsf{TC}_1(\mathsf{sub}^1)(P,Q) \\
&\quad = \mathsf{eq}^1(P,Q) \vee \exists X \left(\mathsf{sub}^1(P,X) \wedge \mathsf{TC}_0(\mathsf{sub}^1)(X,Q)\right) \\
&\quad = \begin{cases} \forall x_1 \left(P(x_1) \leftrightarrow Q(x_1)\right) \vee \\ \quad \exists X \left(\forall x_2 \left(P(x_2) \rightarrow X(x_2)\right) \wedge \mathsf{eq}^1(X,Q)\right) \end{cases} \\
&\quad = \begin{cases} \forall x_1 \left(P(x_1) \leftrightarrow Q(x_1)\right) \vee \\ \quad \exists X \left(\forall x_2 \left(P(x_2) \rightarrow X(x_2)\right) \wedge \forall x_3 \left(X(x_3) \leftrightarrow Q(x_3)\right)\right) \end{cases}
\end{aligned}
$$

In the calculation above, P, Q and X have arity 1.

As the implementation of the combinators is in OCaml this parametrisation is natural and affords us higher-order generality. For example, the OCaml implementation of sub^1 is as follows:

```
let subset a b =
  let y = mk_fresh_fv () in
  FoAll (y, mk_implies [QRel (a, [Var y])] (QRel (b, [Var y])))
```

And the definition of TC_n is as follows:

```
let rec tc n f a b =
  let x = mk_fresh_fv ~prefix:"tc_x" () in
  let step = match n with
      1 -> f
    | _ -> tc (n-1) f
  in
  Or [
    f a b
  ; FoAny (x, And [f a (Var x); step (Var x) b])
  ]
```

The transitive closure of subset is therefore simply `tc n subset`. The relation parameters are curried in the uses of `tc` and `subset`. We use the combinators freely in what follows, and introduce new combinators as convenient.

4.3 SO Solving through QBF

From a reasoning perspective, SO model-checking is a highly non-trivial task due to quantifiers. In particular, quantifiers over relations, where the size of the search-space alone is daunting. For a universe of size n there are 2^{n^2} possible binary relations, and there are 2^{n^k} possible k-ary relations.[2]

A relation is uniquely characterised by a vector of Boolean values, each determining whether a certain tuple is in the relation or not. This insight lets us formulate a search for a relation as a SAT problem, where a fresh Boolean variable is introduced for any potential tuple in the relation. Even though the translation is exponential, it is a popular method in finite-model finding for first-order logic formulae Claessen and Sörensson (2003); Torlak and Jackson (2007); Reger, Suda and Voronkov (2016).

However, in the setting of SO, a SAT solver is insufficient since the input formula may contain alternating quantifiers. We tackle this issue by translating to quantified Boolean formulae (QBF), rather than to plain SAT. The translation is carried out in three stages.

1. each interpreted relation is in-lined as a disjunction of conjunctions over the tuples where the relation holds;

2. first-order quantifiers are expanded into Boolean connectives over the elements of the universe, i.e. $\forall x\phi$ leads to one conjunct for each element of the universe and $\exists x\phi$ leads to one disjunct for each element of the universe;

3. all atoms now are ground and each atom is replaced by a fresh Boolean variable, which is inserted under the same type of quantifier as the atom.

For illustration, consider the formula $\exists X \forall Y \forall z \left(Y(z) \rightarrow X(z)\right)$ and the universe $A = \{1, 2\}$. The formula requires a set X that is a superset of all sets. Inevitably, X has to be the whole domain. The QBF formulation is $\exists \mathsf{x}_1\mathsf{x}_2 \forall \mathsf{y}_1\mathsf{y}_2 \left((\mathsf{y}_1 \rightarrow \mathsf{x}_1) \wedge (\mathsf{y}_2 \rightarrow \mathsf{x}_2)\right)$. Intuitively, rather than talking about a set, we focus on each element separately, which is enabled by the finiteness of the universe. Using QBF enables us to arbitrarily quantify over the sets' elements.

PrideMM enables exporting the QBF formulation into the QCIR format (Jordan, Klieber and Seidl 2016), which is supported by a bevy of QBF solvers. However, since most solvers only support prenex form, PrideMM, also additionally prenexes the formula, where it attempts to heuristically minimise the number of quantifier levels.

[2] Finding constrained finite relations is NEXP-TIME complete (Lewis 1980).

The experimental evaluation showed that the QFUN solver (Janota 2018) performs the best on the considered instances, see section 4.5. While the solver performs very well on the J+R litmus tests, a couple of instances were left unsolved. Encouraged by the success of QFUN, we built a dedicated solver that integrates the translation to QBF and the solving itself. The solver represents the formula in dedicated hash-consed data structures (the formulae grow in size considerably). The expansion of first-order variables is done directly on these data structures while also simplifying the formula on the go. The solver also directly supports non-prenex formulation (see Janota et al. (2016) for non-prenex QBF solving). The solver applies several preprocessing techniques before expanding the first-order variables, such as elimination of relations that appear only in positive or only in negative positions in the formula.

4.4 Memory Specification Encodings

In this section, we show that many memory specifications can be expressed conveniently in second-order logic. We represent programs and their behaviours with event structures: this supports the expression of axiomatic specifications such as C++, but also the higher-order specification of J+R. For a given program, its event structure is constructed in a straightforward way: loads give rise to mutually conflicting read events and writes to write events (Jeffrey and Riely 2016). We express the constraints over event structures with the following vocabulary, shared across all specifications.

Vocabulary. A memory specification decides if a program is allowed to have a certain behaviour. We pose this as a model checking problem, $\mathfrak{A} \models \phi$, where $\mathfrak{A}$ captures program behaviour and ϕ the memory specification. The vocabulary of $\mathfrak{A}$ consists of the following symbols:

- arity 1:`read`, `write`, `final`
- arity 2: $\leq$, `conflict`, `justifies`, `sloc`, $=$

Sets `read` and `write` classify read and write events. The symbol `final`, another set of events, identifies the executions that exhibit final register states matching the outcome specified by the litmus test.

Events x and y are in program order, written $x \leq y$, if event x arises from an earlier statement than y in the program text. We have $\texttt{conflict}(x, y)$ between events that cannot belong to the same execution; for example, a load statement gives rise to an event for each value it might read, but an execution chooses one particular value, and contains only the corresponding event. We

write $\texttt{justifies}(x, y)$ when x is a read and y is a write to the same memory location of the same value. We have $\texttt{sloc}(x, y)$ when x and y access the same memory location. Identity on events, $\{\,(x, x) \mid x \in A\,\}$, is denoted by $=$.

Configurations and Executions. We distinguish two types of sets of events. A *configuration* is a set of events that contains no conflict and is downward closed with respect to $\leq$; that is, X is a configuration when $\mathsf{V}(X)$ holds, where the V combinator is defined by

$$\mathsf{V}(X) \;\coloneqq\; \begin{cases} \forall x \forall y \left(\big(X(x) \wedge X(y)\big) \rightarrow \neg \texttt{conflict}(x, y) \right) \\ \quad \wedge \forall y \left(X(y) \rightarrow \forall x \left((x \leq y) \rightarrow X(x) \right) \right) \end{cases} \tag{1}$$

We say that a configuration X is an *execution of interest* when every final event is either in X or in conflict with an event in X; that is, X is an execution of interest when $\mathsf{F}(X)$ holds, where the F combinator is defined by

$$\mathsf{F}(X) \;\coloneqq\; \mathsf{V}(X) \wedge \forall x \left(\begin{array}{l} \big(\texttt{final}(x) \wedge \neg X(x)\big) \rightarrow \\ \quad \exists y \left(\texttt{conflict}(x, y) \wedge \texttt{final}(y) \wedge X(y) \right) \end{array} \right) \tag{2}$$

Intuitively, we shall put in `final` all the maximal events (according to $\leq$) for which registers have the desired values.

Notations. In the formulae below, X will stand for a configuration, which may be the execution of interest. Variables Y_{rf}, Y_{co}, Y_{hb} and so on are used to represent the relations that are typically denoted by rf, co, hb, … Thus, X has arity 1, while Y_{rf}, Y_{co}, Y_{hb}, … have arity 2.

In what follows, we present four memory specifications: sequential consistency (section 4.4.1), release–acquire (section 4.4.2), C++ (section 4.4.3), and J+R (section 4.4.4). The first three can be expressed in ∃SO (and in first-order logic). The last one uses both universal and existential quantification over sets. For each memory specification, we shall see their encoding in second-order logic.

4.4.1 Sequential Consistency

The SC specification allows all interleavings of threads, and nothing else. In addition to the common relations (`final`, `read`, `write` `conflict`, `justifies`, $\leq$, $=$), the second-order structure $\mathfrak{A}$ also contains the relation `sloc`. We have $\texttt{sloc}(x, y)$ when events x and y denote memory operations on the same location.

It is described by the following SO sentence:

$$\mathsf{SC} \coloneqq \exists X\, Y_{\mathsf{co}}\, Y_{\mathsf{rf}}\, \big(\mathsf{F}(X) \wedge \mathsf{co}(X,\, Y_{\mathsf{co}}) \wedge \mathsf{rf}(X,\, Y_{\mathsf{rf}}) \wedge \mathsf{acyclic}(\mathsf{R}(\,Y_{\mathsf{co}},\, Y_{\mathsf{rf}}))\big)$$

Intuitively, we say that there exists a coherence order relation Y_{co} and a reads-from relation Y_{rf} which, when combined in a certain way, result in an acyclic relation $\mathsf{R}(\,Y_{\mathsf{co}},\, Y_{\mathsf{rf}})$. The formula $\mathsf{co}(X,\, Y_{\mathsf{co}})$ says that Y_{co} satisfies the usual axioms of a coherence order with respect to the execution X; and the formula $\mathsf{rf}(X,\, Y_{\mathsf{rf}})$ says that Y_{rf} satisfies the usual axioms of a reads-from relation with respect to the execution X. Moreover, the formula $\mathsf{F}(X)$ asks that X is an execution of interest, which results in registers having certain values.

$$\mathsf{co}(X,\, Y_{\mathsf{co}}) \coloneqq \begin{cases} \mathsf{trans}(\,Y_{\mathsf{co}}) \wedge \\ \forall xy \left(\begin{array}{l} \big(X(x) \wedge X(y) \wedge \mathtt{write}(x) \wedge \mathtt{write}(y) \wedge \mathtt{sloc}(x,y) \wedge (x \neq y)\big) \\ \quad \leftrightarrow \big(\,Y_{\mathsf{co}}(x,y) \vee\, Y_{\mathsf{co}}(y,x)\big) \end{array} \right) \end{cases} \tag{3}$$

$$\mathsf{rf}(X,\, Y_{\mathsf{rf}}) \coloneqq \begin{cases} \mathsf{inj}(\,Y_{\mathsf{rf}}) \wedge \mathsf{sub}^2(\,Y_{\mathsf{rf}}, \mathtt{justifies}) \wedge \\ \quad \forall y \Big(\big(\mathtt{read}(y) \wedge X(y)\big) \rightarrow \exists x \, \big(\mathtt{write}(x) \wedge X(x) \wedge\, Y_{\mathsf{rf}}(x,y)\big) \Big) \end{cases} \tag{4}$$

When X is a potential execution and Y_{co} is a potential coherence-order relation, the formula $\mathsf{co}(X,\, Y_{\mathsf{co}})$ requires that the writes in X for the same location include some total order. Because of the later condition that $\mathsf{R}(\,Y_{\mathsf{co}},\, Y_{\mathsf{rf}})$ is acyclic, Y_{co} is in fact required to be a total order per location. When X is a potential execution and Y_{rf} is a potential reads-from relation, the formula $\mathsf{rf}(X,\, Y_{\mathsf{rf}})$ requires that Y_{rf} is injective, is a subset of $\mathtt{justifies}$, and relates all the reads in X to some write in X. The helper macros for injectivity and subset, inj and sub^k, are straightforward:

$$\mathsf{inj}(P) \;\coloneqq\; \forall xyz \Big(\big(P(x,z) \wedge P(y,z)\big) \rightarrow (x = y) \Big) \tag{5}$$

$$\mathsf{sub}^1(P,Q) \;\coloneqq\; \forall x \, \big(P(x) \rightarrow Q(x)\big) \tag{6}$$

$$\mathsf{sub}^2(P,Q) \;\coloneqq\; \forall x_1 x_2 \, \big(P(x_1,x_2) \rightarrow Q(x_1,x_2)\big) \tag{7}$$

Note that inj works on binary predicates, while sub^k works on predicates of arity k.

The auxiliary relation $\mathsf{R}(\,Y_{\mathsf{co}},\, Y_{\mathsf{rf}})$ is the union of strict program-order $(<)$, reads-from $(\,Y_{\mathsf{rf}})$, coherence-order $(\,Y_{\mathsf{co}})$, and the from-reads relation:

$$\mathsf{R}(\,Y_{\mathsf{co}},\, Y_{\mathsf{rf}})(y,z) \;\coloneqq\; (y < z) \vee\, Y_{\mathsf{co}}(y,z) \vee\, Y_{\mathsf{rf}}(y,z) \vee \exists x \, \big(\,Y_{\mathsf{co}}(x,z) \wedge\, Y_{\mathsf{rf}}(x,y)\big) \tag{8}$$

The acyclicity check is implemented as follows:

$$\mathsf{acyclic}(P) \;\coloneqq\; \exists X \left(\mathsf{sub}^2(P, X) \wedge \mathsf{sub}^2(\mathsf{seq}(X, X), X) \wedge \forall x\, \neg X(x, x)\right) \tag{9}$$

In words, P is acyclic if it is contained in some X that is transitive and irreflexive. There exist many other encodings for acyclicity checks, but this one leads to good performance in practice with current SAT solvers (Janota, Grigore and Manquinho 2017). The helper macro seq is defined as follows:

$$\mathsf{seq}(P, Q)(x, z) \;\coloneqq\; \exists y \left(P(x, y) \wedge Q(y, z)\right) \tag{10}$$

4.4.2 Release–Acquire

Release–Acquire is a simple relaxed memory specification, which is represented straightforwardly in SO logic. It is captured by the formula RA using the vocabulary established in the definition of SC:

$$\mathsf{RA} \;\coloneqq\; \exists X\, Y_{\mathsf{co}}\, Y_{\mathsf{rf}} \left(\begin{array}{l} \mathsf{F}(X) \wedge \mathsf{co}(X, Y_{\mathsf{co}}) \wedge \mathsf{rf}(X, Y_{\mathsf{rf}}) \wedge \mathsf{acyclic}(Y_{\mathsf{co}}) \\ \quad \wedge \exists Y_{\mathsf{hb}} \left(\begin{array}{l} \mathsf{sub}^2(<, Y_{\mathsf{hb}}) \wedge \mathsf{sub}^2(Y_{\mathsf{rf}}, Y_{\mathsf{hb}}) \wedge \mathsf{trans}(Y_{\mathsf{hb}}) \\ \quad \wedge \mathsf{irrefl}(Y_{\mathsf{hb}}) \wedge \mathsf{irrefl}(\mathsf{seq}(Y_{\mathsf{co}}, Y_{\mathsf{hb}})) \\ \quad \wedge \mathsf{irrefl}(\mathsf{seq}(\mathsf{inv}(Y_{\mathsf{rf}}), \mathsf{seq}(Y_{\mathsf{co}}, Y_{\mathsf{hb}}))) \end{array} \right) \end{array} \right) \tag{11}$$

The existential SO variable Y_{hb} over-approximates a relation traditionally called happens-before.

4.4.3 C++

To capture the C++ specification in SO logic, we follow the Herd7 specification of Lahav et al. (2017). Their work introduces necessary patches to the specification of the standard (Batty et al. 2011) but also includes fixes and adjustments from prior work (Batty, Donaldson and Wickerson 2016; Lahav, Giannarakis and Vafeiadis 2016). The specification is more nuanced than the SC and RA specifications and requires additions to the vocabulary of $\mathfrak{A}$ together with a reformulation for efficiency, but the key difference is more fundamental. C++ is a *catch-fire* semantics: programs that exhibit even a single execution with a data race are allowed to do

anything — satisfying every expected outcome. This difference is neatly expressed in SO logic:

$$\mathsf{CPP} \coloneqq \exists X\, Y_{\mathsf{co}}\, Y_{\mathsf{rf}}\, Y_{\alpha\beta} \begin{pmatrix} \mathsf{co}(X, Y_{\mathsf{co}}) \wedge \mathsf{rf}(X, Y_{\mathsf{rf}}) \wedge \mathsf{hb}(Y_{\alpha\beta}, Y_{\mathsf{rf}}) \\ \wedge\, \mathsf{M}(Y_{\alpha\beta}, Y_{\mathsf{co}}, Y_{\mathsf{rf}}) \wedge (\mathsf{F}(X) \vee \mathsf{C}(Y_{\alpha\beta}, Y_{\mathsf{rf}})) \end{pmatrix} \tag{12}$$

The formula reuses $\mathsf{co}(X, Y_{\mathsf{co}})$, $\mathsf{rf}(X, Y_{\mathsf{rf}})$, and $\mathsf{F}(X)$ and includes three new combinators: $\mathsf{hb}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$, $\mathsf{M}(Y_{\alpha\beta}, Y_{\mathsf{co}}, Y_{\mathsf{rf}})$ and $\mathsf{C}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$. $\mathsf{hb}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$ constrains a new over-approximation, $Y_{\alpha\beta}$, used for building a transitive relation. $\mathsf{M}(Y_{\alpha\beta}, Y_{\mathsf{co}}, Y_{\mathsf{rf}})$ captures the conditions imposed on a valid C++ execution, and is the analogue of the conditions applied in SC and RA. $\mathsf{C}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$ holds if there is a race in the execution X. Note that the expected outcome is allowed if $\mathsf{F}(X)$ is satisfied or if there is a race and $\mathsf{C}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$ is true, matching the catch-fire semantics.

New vocabulary. C++ *Read-modify-write* operations load and store from memory in a single atomic step: a new rmw relation links the corresponding reads and writes. C++ *fence* operations introduce new events and the set fences identifies them. The programmer annotates each memory access and fence with a *memory order* parameter that sets the force of inter-thread synchronisation created by the access. For each choice, we add a new set: na, rlx, acq, rel, acq_rel, and sc.

Over-approximation in happens before. The validity condition, $\mathsf{M}(Y_{\alpha\beta}, Y_{\mathsf{co}}, Y_{\mathsf{rf}})$, and races $\mathsf{C}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$, hinge on a relation called *happens-before*. We over-approximate transitive closures in the SO logic for efficiency, but Lahav et al. (2017) define happens-before with nested closures that do not perform well. Instead, we over-approximate a reformulation of happens-before that flattens the nested closures into a single one.

We define a combinator for happens-before, $\mathsf{HB}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$, that is used in $\mathsf{M}(Y_{\alpha\beta}, Y_{\mathsf{co}}, Y_{\mathsf{rf}})$ and $\mathsf{C}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$. It takes as argument an over-approximation of the closure internal to the reformed definition of happens-before, $Y_{\alpha\beta}$. $\mathsf{hb}(Y_{\alpha\beta}, Y_{\mathsf{rf}})$ constrains $Y_{\alpha\beta}$, requiring it to be transitive and to include the conjuncts of the closure, α and β below.

$$\mathsf{HB}(Y_{\alpha\beta}, Y_{\mathsf{rf}}) \coloneqq \mathsf{or}(<, \mathsf{seq}(\mathsf{maybe}(<), \mathsf{sw}_{\mathsf{begin}}(Y_{\mathsf{rf}}), Y_{\alpha\beta}, \mathsf{sw}_{\mathsf{end}}(Y_{\mathsf{rf}}), \mathsf{maybe}(<))) \tag{13}$$

$$\alpha(Y_{\mathsf{rf}}) \coloneqq \mathsf{seq}(\mathsf{sw}_{\mathsf{end}}(Y_{\mathsf{rf}}), \mathsf{maybe}(<), \mathsf{sw}_{\mathsf{begin}}(Y_{\mathsf{rf}})) \tag{14}$$

$$\beta(Y_{\mathsf{rf}}) \coloneqq \mathsf{seq}(Y_{\mathsf{rf}}, \mathsf{rmw}) \tag{15}$$

$$\mathsf{hb}(Y_{\alpha\beta}, Y_{\mathsf{rf}}) \coloneqq \begin{cases} \mathsf{trans}(Y_{\alpha\beta}) \\ \wedge\, \mathsf{sub}^2(\mathsf{id}, Y_{\alpha\beta}) \wedge \mathsf{sub}^2(\alpha(Y_{\mathsf{rf}}), Y_{\alpha\beta}) \wedge \mathsf{sub}^2(\beta(Y_{\mathsf{rf}}), Y_{\alpha\beta}) \end{cases} \tag{16}$$

4.4.4 Jeffrey–Riely

The J+R memory specification is captured by a sentence JR_n, parametrised by an integer n. Unlike the formulae we saw before, JR_n makes use of three levels of quantifiers ($\exists\forall\exists$), putting it on the third level of the polynomial hierarchy. We begin by lifting[3] `justifies` from events to sets of events P and Q:

$$\mathsf{J}(P,Q) \coloneqq \forall y \left(\begin{array}{l} \big(\neg P(y) \wedge Q(y) \wedge \mathtt{read}(y)\big) \\ \quad \rightarrow \exists x \left(P(x) \wedge \mathtt{write}(y) \wedge \mathtt{justifies}(x,y)\right) \end{array} \right) \tag{17}$$

$$\mathsf{AJ}(P,Q) \coloneqq \mathsf{J}(P,Q) \wedge \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \tag{18}$$

We read J as 'justifies', and AJ as 'always justifies'. Next, we define what Jeffrey and Riely call 'always eventually justify'

$$\mathsf{AeJ}_n(P,Q) \coloneqq \begin{cases} \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \wedge \\ \forall X \left(\mathsf{TC}_n(\mathsf{AJ})(P,X) \rightarrow \exists Y \left(\mathsf{TC}_n(\mathsf{AJ})(X,Y) \wedge \mathsf{J}(Y,Q) \right) \right) \end{cases} \tag{19}$$

The size of the formula $\mathsf{TC}_n(\mathsf{AeJ}_m)(P,Q)$ we defined above is $\Theta(mn)$. In particular, it is bounded. Finally, we let[4]

$$\mathsf{JR}_n \coloneqq \exists X \left(\mathsf{TC}_n(\mathsf{AeJ}_n)(\emptyset, X) \wedge \mathsf{F}(X) \right) \tag{20}$$

and ask solve the model checking problem $\mathfrak{A} \models \mathsf{JR}_n$. Since the formulae above are in MSO, it is sufficient to pick $n \coloneqq 2^{|A|}$. Since all bounded transitive closures include the subset relation, they are monotonic, and it suffices, in fact, to pick $n \coloneqq |A|$. For actual solving, we will use this observation.

4.5 Evaluation

We evaluate our tool in the context of Herd7 (Alglave, Maranget and Tautschnig 2014), which is a standard tool among memory specification researchers for building axiomatic memory specifications. No similar tool exists for higher-order event structure based memory specifications.

[3] Our definition of J is different from the original one (Jeffrey and Riely 2016): we require that only new reads are justified, by including the conjunct $\neg P(y)$. Without this modification, our solver's results disagree with the hand-calculations reported by Jeffrey and Riely; with this modification, the results agree.

[4] The symbol $\emptyset$ denotes the empty unary relation, as expected.

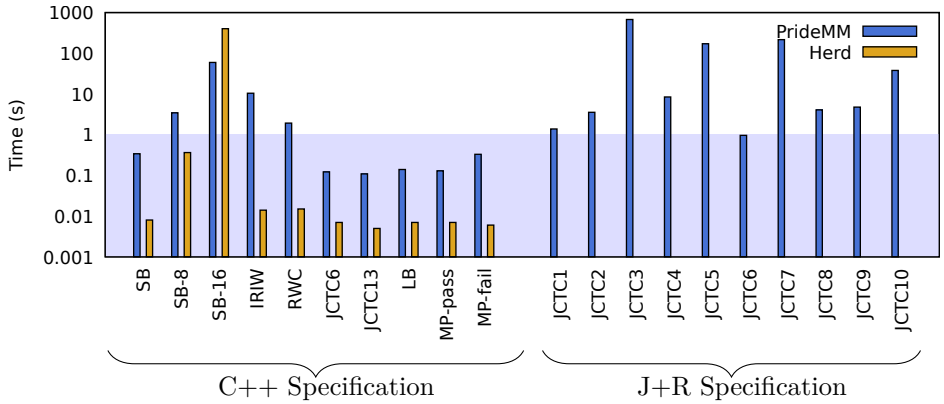


Figure 43: Comparison of PrideMM's performance in contrast to Herd7.

4.5.1 Comparison to existing techniques

In figure fig. 43 we compare the performance and capabilities of PrideMM to Herd7, the de facto standard tool for building axiomatic memory specifications. Herd7 and PrideMM were both executed on a machine equipped with an Intel i5-5250u CPU and 16 GB of memory. We choose not to compare our tool to MemSAT (Torlak, Vaziri and Dolby 2010), as there are more memory specifications implemented for Herd7 in the CAT language (Alglave, Cousot and Maranget 2016) than there are for MemSAT. Herd is also a good point of comparison because of its industrial and academic success, with ARM's canonical memory specification being described in Herd (Deacon and Alglave 2019).

Performance. Notably Herd7's performance is very favourable in contrast to the performance of PrideMM, however there are some caveats. The performance of PrideMM is largely adequate, with most of the standard litmus tests taking less than 2 seconds to execute. $y \leq 1s$ is highlighted on the chart. We find that our QBF technique scales better than Herd7 with large programs. This is demonstrated in the SB-16 test, a variant of the "store buffering" litmus test with 16 threads. The large number of combinations for quantifying the existentially quantified relations which are explored naïvely by Herd7 cause it to take a long time to complete. In contrast, smarter SAT techniques handle these larger problems handily.

Expressiveness. We split the chart in figure fig. 43 into 2 sections, the left-hand side of the chart displays a representative subset of common litmus tests showing PrideMM's strength and weaknesses. These litmus tests are evaluated under the C++ memory specification. Note that these include tests with behaviour expected to be observable and unobservable, hence there being

Prob.	SAT	caqe (s)	qfun (s)	qfm (s)
1	N	⊥	610	**2**
2	N	⊥	23	**2**
3	Y	⊥	⊥	**222**
4	Y	⊥	**2**	5
5	Y	⊥	78	**51**
6	N	5	4	**1**
7	Y	⊥	280	**56**
8	N	⊥	**2**	**2**
9	N	⊥	2	**1**

Prob.	SAT	caqe (s)	qfun (s)	qfm (s)
10	Y	⊥	36	**10**
11	Y	⊥	598	**335**
13	Y	**1**	**1**	**1**
14	Y	⊥	**29**	33
15	Y	⊥	512	**157**
16	N	⊥	⊥	**12**
17	N	⊥	**39**	311
18	N	⊥	359	**190**
#17		#2	#15	#17

Figure 44: Solver approaches for PrideMM on Java Causality Test Cases. ⊥ represents timeout or mem-out.

two MP bars. The C++ memory specification is within the domain of memory specifications that Herd7 can solve, as it requires only existentially quantified relations.

The right-hand half of the chart is the first 10 Java causality test cases run under the J+R specification, which are no longer expressible in Herd7. PrideMM solves these in reasonable time, with most tests solved in less than 10 minutes. Our J+R tests replicate the results found in the original paper, but where they use laborious manual proof in the Agda proof assistant, PrideMM validates the results automatically.

4.5.2 QBF vs SO Solver Performance

PrideMM enables emitting the SO logic formulae and structures directly for the SO solver, or we can convert to a QBF query (see section 4.3). This allows us to use our SO solver as well as QBF solvers. We find that the SO solver affords us a performance advantage over the QBF solver in most of the Java causality test cases, where performance optimisations for alternating quantification are applicable.

We include the performance of the QBF solvers CAQE and QFUN, the respective winners of the CNF and non-CNF tracks at 2017's QBFEVAL competition (QBFLIB 2017). Our QBF benchmarks were first produced in the circuit-like format QCIR (Jordan, Klieber and Seidl 2016), natively supported by QFUN. The inputs to CAQE were produced by converting to CNF through standard means, followed by a preprocessing step with Bloqqer (Biere, Lonsing and Seidl 2011).

We can also emit the structures and formulae as an Isabelle/HOL file, which can then be loaded into Nitpick (Blanchette and Nipkow 2010) conveniently. We found that Nitpick cannot be run over the C++ specification or the J+R specification, timing out after 1 hr on all the litmus tests.

4.6 Related Work

We build on prior work from two different areas — relaxed memory specifications, and SAT/QBF solving: the LISA frontend comes from the Herd7 memory-specification simulator (Alglave, Maranget and Tautschnig 2014), the MMGs implement memory specifications that have been previously proposed (Lahav et al. 2017; Jeffrey and Riely 2016), and the SO solver is based on a state-of-the-art QBF solver (Janota 2018).

There is a large body of work on finite relational model finding in the context of memory specifications using Alloy (Jackson 2002). Alloy has been used to compare memory specifications and find litmus tests which can distinguish two specifications (Wickerson et al. 2017), and has been used to synthesise comprehensive sets of tests for a specific memory specification (Lustig et al. 2017). Applying SAT technology in the domain of evaluating memory specifications has been tried before, too. MemSAT (Torlak, Vaziri and Dolby 2010) uses Kodkod (Torlak and Jackson 2007), the same tool that Alloy relies on to do relational model finding. MemSynth (Bornholt and Torlak 2017b) uses Ocelot (Bornholt and Torlak 2017a) to embed relational logic into the Rosette (Torlak and Bodik 2014) language. Our results are consistent with the findings of MemSAT and MemSynth: SAT appears to be a scalable and fast way to evaluate large memory specification questions. Despite this, SAT does not widen the class of specifications that can be efficiently simulated beyond ad hoc techniques.

There is work to produce a version of Alloy which can model higher-order constructions, called Alloy* (Milicevic et al. 2015), however this is limited in that each higher order set requires a new signature in the universe to represent it. Exponential expansion of the sets quantified in the J+R specification leaves model finding for J+R executions intractable in Alloy* too.

While Nitpick (Blanchette and Nipkow 2010) can model higher order constructions, we found it could not generate counter examples in a reasonable length of time of the specifications we built. There is work to build a successor to Nitpick called Nunchaku (Reynolds et al. 2016), however, at present Nunchaku does not support higher order quantification. Once Nunchaku is more complete we intend to output to Nunchaku and evaluate its performance in comparison to our SO solver.

There is a bevy of work on finite model finding in various domains. SAT is a popular method for finite model finding in first-order logic formulae (Claessen and Sörensson 2003; Reger, Suda and Voronkov 2016). There are constraint satisfaction-based model finders, e.g. the SEM model finder (Zhang and Zhang 1995), relying on dedicated symmetry and propagation. Reynolds et al. propose solutions for finite model finding in the context of SMT (Reynolds et al. 2013a,b) (CVC4 is in fact used as backend to Nunchaku).

4.7 Conclusion

This chapter presented PrideMM, a case study of the application of new solving techniques to a problem domain with active research. PrideMM allows memory specification researchers to build a new class of memory specifications with richer quantification, and still automatically evaluate these specifications over programs. In general, for an axiomatic model Herd7 remains the right tool for the job: it is faster, and has a mature ecosystem. For a higher-order model like J+R however, applying solvers in the style of PrideMM allows a direct encoding of the memory model without error-prone implementation of underlying mathematical constructions. In this sense we provide a Herd7-style modify-execute-evaluate pattern of development for higher-order memory specifications that were previously unsuitable for mechanised model finding.

Chapter 5

Computing Thin-Air Free Denotations

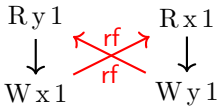
In this chapter we define Modular Relaxed Dependencies: a thin-air free semantics for weak memory concurrency which can be extended to cover C/C++11's features, and is amenable to automatic evaluation. This work was done with collaborators: Marco Paviotti, Daniel Wright, Anouk Paradis, Scott Owens, and Mark Batty. My contributions to this work are significant: the definition of coproduct (§5.4.2) – which is core to our semantics – is the result of careful iteration between tool building and mathematical formulation. Without this iteration construction of a tool would have been impractical, and hand evaluation of coproduct would have been more arduous and error prone.

5.1 The problem with program dependencies

Program dependencies are hard to precisely capture in a programming language semantics. It would be desirable to have a notion of *semantic dependency* (sdep) (McKenney et al. 2016). Semantic dependencies are dependencies which are preserved through compiler optimisations and are also preserved by the semantics of the machine the program will execute on. Our work on Modular Relaxed Dependencies provides a semantic dependency relation which can be used with axiomatic memory models. It is calculated by a *denotational* semantics of programs, where the final denotation can be projected into a set of executions and dependency relations for direct use with an axiomatic model. The result is a thin-air free semantics without the overly strong restriction that $(\mathsf{rf} \cup \mathsf{po})$ is acyclic.

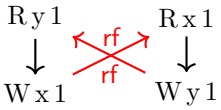
As in §2.3, we will consider how syntactic dependencies do not provide an adequate description of the dependencies which will be later present in the optimised compiler generated code. First, let us re-examine load buffering.

Initially: x := 0; y := 0;	
r_1 := y;	r_2 := x;
x := 1	y := 1

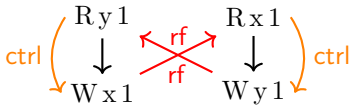


It is important that a concurrency semantics for C/C++ permits this weak execution where $r_1 = r_2 = 1$. Compiler optimisations (and hardware optimisations) must be permitted to re-order the loads and stores on each thread, unfortunately in the axiomatic setting, weak executions of the program above are indistinguishable from the following program: LB+dep.

Initially: x := 0; y := 0;	
r_1 := y;	r_2 := x;
if($r_1 \neq 0$){x := 1}	**if**($r_2 \neq 0$){y := 1}

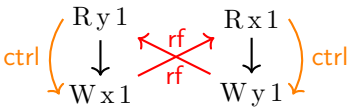


One might consider annotating the executions with syntactic dependencies – *i.e.* those dependencies which are present in the source code – and then simply forbid cycles in ctrl ∪ rf. Such an execution is printed below, with a pair of additional ctrl edges.



Unfortunately this is not good enough, as simple modification to the program to introduce false dependencies will show.

Initially: x := 0; y := 0;	
r_1 := y;	r_2 := x;
if($r_1 \neq 0$) {	**if**($r_2 \neq 0$) {
x := 1	y := 1
} **else** {	} **else** {
x := 1	y := 1
}	}

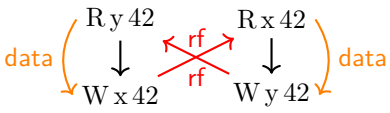


In LB+false-dep (above) we see a program which can be trivially optimised to the original LB program. We also show the weak execution anotated with the syntactic dependency, in this case the control dependencies (ctrl). This optimisation is allowed by modern compilers, and preserving permitted optimisations is an important goal in weak memory consistency. Thus, as we want to allow these optimisations, so too must we allow the weak execution. Forbidding a cycle in rf ∪ ctrl is therefore inadequate. One final example is LB+datas.

Initially: `x := 0; y := 0;`

r_1 := y;	r_2 := x;
x := r_1	y := r_2

Here we see that the value loaded is immediately stored, which leads to an interesting problem. There are real data dependencies in this program, but if we do not outlaw cycles in data ∪ rf, then we are forced to allow the following nonsensical execution.



This is problematic: if one wishes to prove safety properties about a C-program, like for example that `x` never points to address 42, then the current axiomatic memory model of C/C++ will not help – we can write a program with an allowed execution which summons the value 42 out-of-thin-air. C and C++ deliberately allow the optimisation of LB+false-dep to LB, and other similar optimisations which remove syntactic dependencies, and the language model must disregard these potentially unpreserved dependencies to be agnostic to compiler optimisation. Herein lies the problem with axiomatic memory models: to forbid OOTA behaviour we must find the dependencies which are not just syntactic, but semantic. McKenney et al. (2016) quotes an observation that Ali Sezgin made: a semantic dependency relation (sdep) would be an ideal solution – *"Prohibiting executions that have cycles in* rf ∪ sdep *can therefore be expected to prohibit OOTA behaviors."*

5.1.1 Modular Relaxed Dependency by example

To simplify things for now, we will attach an `Init` program to the beginning of each example to initialise all global variables to zero. Doing this makes the semantics non-compositional, but it is

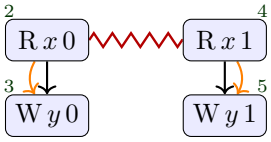
a natural starting place and aligns well with previous work in the area. Later, after we have made all of our formal definitions, we will see why the `Init` program is not necessary.

For now, consider a simple programming language where all values are booleans, registers (ranging over `r`) are thread-local, and variables (ranging over `x, y`) are global. Informally, an event structure for a program consists of a directed graph of events. Events represent the global variable reads and writes that occur on all possible paths that the program can take. This can be built up over the program as follows: each write generates a single event, while each read generates two – one for each possible value that could be read. These read events are put in *conflict* with each other to indicate that they cannot both happen in a single execution, this is indicated with a zig-zag red arrow between the two events. Additionally, the event structure tracks true dependencies via an additional relation which we call *semantic dependencies* (sdep). These are yellow arrows from read events to write events.

For example, consider the program

$$(\texttt{r}_1 \texttt{ := x; y := r}_1) \tag{LB_1}$$

that reads from a variable `x` and then writes the result to `y`. The interpretation of this program is an event structure depicted as follows:



Each event has a unique identifier (the number attached to the box). The straight black arrows represent program order, the curved yellow arrows indicate a causal dependency between the reads and writes, and the red zigzag represents a conflict between two events. If two events are in conflict, then their respective continuations are in conflict too.

If we interpret the program `Init;` LB_1, as below, we get a program where the `Init` event sets the variables to zero.

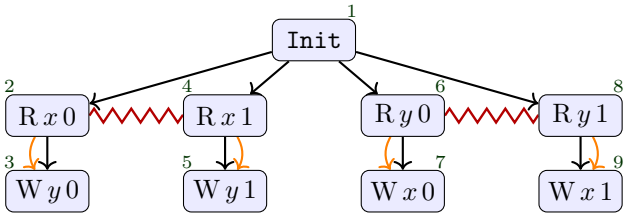
In the above event structure, we highlight events $\{1, 2, 3\}$ to identify an execution. The green dotted arrow indicates that event 2 reads its value from event 1, we call this relation *reads-from* (rf). This execution is *complete* as all of its reads read from a write and it is closed w.r.t conflict-free program order.

We interpret the following program similarly,

$$(\texttt{r}_2 \texttt{ := y; x := r}_2) \tag{LB_2}$$

leading to a symmetrical event structure where the write to `x` is dependent on the read from `y`.

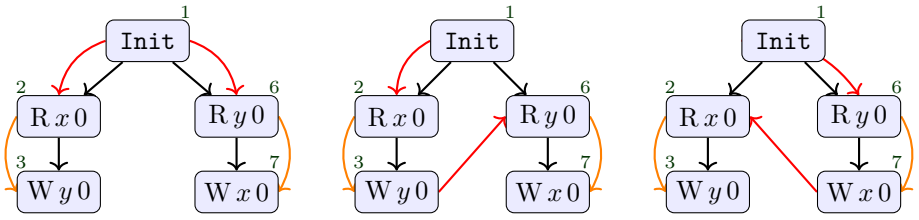
The interpretation of `Init; (`LB_1 `||` LB_2`)` gives the event structure where (LB_1) and (LB_2) are simply placed alongside one another.



The interpretation of parallel composition is the union of the event structures from LB_1 and LB_2 without any additional conflict edges. When parallel composing the semantics of two programs, we add all rf-edges that satisfy a coherence axiom. Here we present an axiom that provides desirable behaviour in this example (Section 5.4 provides our model's complete axioms).

$$(\mathsf{sdep} \cup \mathsf{rf}) \text{ is acyclic}$$

The program `Init; (`LB_1 `||` LB_2`)` allows executions of the following three shapes.

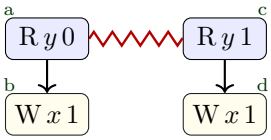


Note that in this example, we are not allowed to read the value 1 – reading a value that does not appear in the program is one sort of thin-air behaviour, as described by Batty et al. (2015). For example, the execution $\{1, 4, 5, 8, 9\}$ does not satisfy the coherence axiom as $4 \xrightarrow{\mathsf{sdep}} 5 \xrightarrow{\mathsf{rf}} 8 \xrightarrow{\mathsf{sdep}} 9 \xrightarrow{\mathsf{rf}} 4$ forms a cycle in $(\mathsf{sdep} \cup \mathsf{rf})$.

We now substitute (LB_2) with the following code snippet

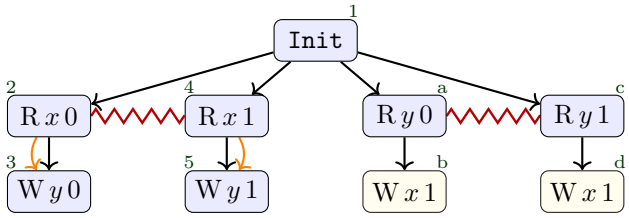
$$\mathtt{r}_1 \mathtt{ := y; x := } 1 \qquad (LB_3)$$

where the value written to the variable `x` is a constant. Its generated event structure is depicted as follows

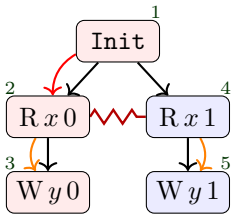


In this program, for each branch, we can reach a write of value 1 to location `x`. Hence, this will happen no matter which branch is chosen: we say b and d are *independent writes* and we draw no dependency edges from their preceding reads.

Consider now the program (LB_3) in parallel with LB_1 introduced earlier in this section. As usual, we interpret the `Init` program in sequence with $(LB_1 \parallel LB_3)$ as follows:



The resulting event structure is very similar to that of $(LB_1 \parallel LB_2)$, but the executions permitted in this event structure are different. The dependency edges calculated when adding the read are preserved, and now executions $\{1, 2, 3, a, b\}$ and $\{1, a, b, 4, 5\}$ are allowed. However, this event structure also contains the execution in which d is independent.



In this execution there is no rf or sdep edge between d and c that can create a cycle, hence this is a valid complete execution in which we can observe $\mathtt{x} = 1, \mathtt{y} = 1$. Note that the `Init` event is irrelevant in the consistency of this execution.

Modularity. It is worthwhile underlining the role that modularity plays here. In order to compute the behaviour of $(LB_1 \parallel LB_2)$ and $(LB_1 \parallel LB_3)$ we did not have to compute the behaviour of LB_1 again. In fact, we computed the semantics of LB_1, LB_2 and LB_3 in isolation and then we observed the behaviour in parallel composition.

Thin-air values. The program $(LB_1 \parallel LB_3)$ is a standard example in the weak memory literature called *load buffering*. In the program $(LB_1 \parallel LB_2)$, if event 5 or 9 were allowed in a complete execution, that would be an undesirable thin-air behaviour: there is no value 1 in the program text, nor does any operation in the program compute the value 1. The program $(LB_1 \parallel LB_3)$ is similar, but now contains a write of value 1 in the program text, so this is no longer a thin-air value. Note that the execution given for it is not sequentially consistent, but

nonetheless a weak memory model needs to allow it so that a compiler can, for example, swap the order of the two commands in LB_3, which are completely independent of each other from its perspective.

5.2 Event Structures

In Chapter 4 we saw how event structures can be used to describe the possible executions of concurrent programs. In this chapter, event structures are used to the same effect, but we find that a step-wise calculation can be performed to explore the possible program states. We will use this calculation to find branches of program execution which can always be reached, referring back to McKenney's C++ paper: we will detect dependency relationships which result in at least some changes in the dynamic execution of programs.

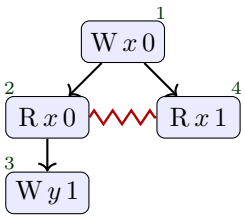
5.2.1 Event Structure Definitions

A partial order $(E, \sqsubseteq)$ is a set E equipped with a reflexive, transitive and antisymmetric relation $\sqsubseteq$. A well-founded partial order is a partial order that has no infinite decreasing chains of the form $\cdots \sqsubseteq e_{i-1} \sqsubseteq e_i \sqsubseteq e_{i+1} \cdots$.

A *prime event structure* is a triple $(E, \sqsubseteq, \#)$. E is a set of events, $\sqsubseteq$ is a well-founded partial order on E and $\#$ is a conflict relation on E. $\#$ is binary, symmetric and irreflexive such that, for all $c, d, e \in E$, if $c\#d \sqsubseteq e$ then $c\#e$. We write $\mathsf{Con}(E)$ for the set of *conflict-free* subsets of E, i.e. those subsets $C \subseteq E$ for which there is no $c, d \in C$ such that $c\#d$.

Notation. We use E to range over (prime/labelled/memory) event structures, and also the event set contained within, when there is no ambiguity. We also use $\mathcal{E}$ for event structures.

A *labelled event structure* $(E, \sqsubseteq, \#, \lambda)$, over a set of labels Σ, is a prime event structure together with a function $\lambda : E \to \Sigma$ which assigns a label to an event. We make events explicit using the notation $\{e : \sigma\}$ for $\lambda(e) = \sigma$. We sometimes avoid using names and just write the label σ when there is no risk of confusion.



Consider the labelled event structure formed by the set $\{1, 2, 3, 4\}$, where the order relation is defined such that $1 \sqsubseteq 2 \sqsubseteq 3$ and $1 \sqsubseteq 4$, the conflict relation is defined such that $2\#4$ and $3\#4$, and the labelling function is defined such that $\lambda(1) = (\mathrm{W}\, x\, 0)$, $\lambda(2) = (\mathrm{R}\, x\, 0)$, $\lambda(3) = (\mathrm{W}\, y\, 1)$ and $\lambda(4) = (\mathrm{R}\, x\, 1)$. The event structure is visualised above. As in Chapter 4, we elide conflict edges that can be inferred from order.

Given labelled event structures $\mathcal{E}_1$ and $\mathcal{E}_2$ define the *product* labelled event structure $\mathcal{E}_1 \times \mathcal{E}_2 \triangleq (E, \sqsubseteq, \#, \lambda)$. E is $E_1 \cup E_2$, assuming E_1 and E_2 to be disjoint, $\sqsubseteq$ is $\sqsubseteq_1 \cup \sqsubseteq_2$, $\#$ is $\#_1 \cup \#_2$ and λ is $\lambda_1 \cup \lambda_2$.

The *coproduct* labelled event structure $\mathcal{E}_1 + \mathcal{E}_2$ is the same as the product, except that the conflict relation $\#$ is $\#_1 \cup \#_2 \cup \{E_1 \times E_2\} \cup \{E_2 \times E_1\}$. We can use a similar construction for the coproduct of an infinite set of pairwise-disjoint labelled event structures, indexed by I: we take infinite unions on the underlying sets and relations, along with extra conflicts for every pair of indices. Where the $\mathcal{E}_i$ are not disjoint, we can make them so by renaming with fresh event identifiers. In particular, we will need the infinite coproduct $\sum_{i \in I} \mathcal{E}$ with as many copies of $\mathcal{E}$ as the cardinality of the set I, and all the events between each copy in conflict. Each of these copies will by referred to as $\mathcal{E}^i$.

For a labelled event structure $\mathcal{E}_0$ and an event e, where $e \notin E_0$, define the *prefix* labelled event structure, $e \bullet \mathcal{E}_0$, as a labelled event structure $(E, \sqsubseteq, \#, \lambda)$ where E equals $E_0 \cup \{e\}$, $\sqsubseteq$ equals $\sqsubseteq_0 \cup (\{e\} \times E)$, and $\#$ equals $\#_0$.

5.3 Coherent event structure

To rigorously define the coherent event structures model, we begin with the labels.

Definition 1 (The signature of labels)**.**

$$\Sigma = (\{R, W\} \times \mathcal{X} \times \mathcal{V}) + \{L\} + \{U\}$$

where $(\mathrm{W}\, x\, v) \in \Sigma$ and $(\mathrm{R}\, x\, v) \in \Sigma$ are the usual write and read operations and L, U are the lock and unlock operations respectively.

A *coherent event structure* is a tuple $(E, S, \vdash, \leq)$ where E is a labelled event structure. S is a set of *partial executions*, where each execution is a tuple comprising a maximal conflict-free set of events, together with an intra-thread reads-from relation rf_i, an extra-thread reads-from rf_e, a semantic dependency relation sdep, and a *partial order* on lock/unlock events lk. The justification relation, $\vdash$, is a relation between conflict-free sets and events. Finally, the *preserved*

program order, $\leq^{\mathcal{X}}$, is a restriction of the program order, $\sqsubseteq$, for events on the same variable. $\leq^{\mathsf{L}}$ is the restriction of program order on events related in program order with locks or unlocks. Finally, we define rf to be $\mathsf{rf}_e \cup \mathsf{rf}_i$ and $\leq$ to be $\leq^{\mathcal{X}} \cup \leq^{\mathsf{L}}$. For a partial execution, $X \in S$, we denote its components as lk_X, rf_X and sdep_X.

Justification, $\vdash$, collects dependency information in the program and is used to calculate sdep_X. For a conflict-free set C and an event e, we say C *justifies* e or e *depends* on C whenever $C \vdash e$. We collect dependencies between events modularly in order to identify the so-called independent writes which will be introduced shortly.

For a given partial execution, X, we define the order hb_X as the reflexive transitive closure of $(\sqsubseteq \cup \mathsf{lk}_X)$. A coherent event structure contains a *data race* if there exists an execution X, with two events on the same variable x, at least one of which is a write, that are not ordered by hb_X. A coherent event structure is *data-race-free* if it does not contain any data race. A *racy* rf*-edge* is when two events w and r are racy and $w \xrightarrow{\mathsf{rf}_{e\,X}} r$. Note that rf_i edges cannot ever be racy.

We now define a coherent partial execution.

Definition 2 (Coherent Partial Execution)**.** *A partial execution X is* coherent *if and only if:*

1. $(\leq^{\mathsf{L}} \cup \mathsf{lk}_X \cup \mathsf{sdep}_X \cup \mathsf{rf}_{e\,X})$ *is acyclic, and*
2. *if* $(w : \mathrm{W}\,x\,v) \xrightarrow{\mathsf{rf}_X} (r : \mathrm{R}\,x\,v)$ *there are no* $(e : \mathrm{R}\,x\,v')$ *or* $(e : \mathrm{W}\,x\,\_)$ *such that* $w \xrightarrow{\mathsf{hb}_X} e \xrightarrow{\mathsf{hb}_X} r$ *with* $v \neq v'$.

5.4 Memory event structures

Central to the model is the way it records program dependencies in $\vdash$ and sdep. Justification, $\vdash$, records the structure of those dependencies in the program that may be influenced by further composition. As we shall see, composing programs may add or remove dependencies from justification: for example, composing a read may make later writes dependent, or the coproduct mechanism, introduced shortly, may remove them. In some parts of the program, e.g. inside locked regions, dependencies do not interact with the context. In this case, we *freeze* the justifications, using them to calculate sdep. Following a freeze, the justification relation is redundant and can be forgotten – sdep can be used to judge which executions are coherent.

Freezing. Here we define a function *freeze* which takes a justification $C \vdash (w : \mathrm{W}\,x\,v)$ and gives the corresponding dependency relation $(r : \mathrm{R}\,x\,v) \xrightarrow{\mathsf{sdep}} (w : \mathrm{W}\,x\,v)$ iff $r \in C$. We lift *freeze* to a function on an event structure as follows:

$$\mathit{freeze}(E_1, S_1, \vdash_1, \leq_1) \triangleq (E_1, S, \emptyset, \leq_1)$$

where S contains all the executions

$$(X_1, \mathsf{lk}_{X_1}, (\mathsf{sdep}_{X_1} \cup \mathsf{sdep}), \mathsf{rf}_{X_1})$$

where for each write, $w_i \in X_1$, we choose a justification so that $C_1 \vdash_1 w_1, ..., C_n \vdash_1 w_n$ covers all writes in X_1. Furthermore, with sdep defined as follows:

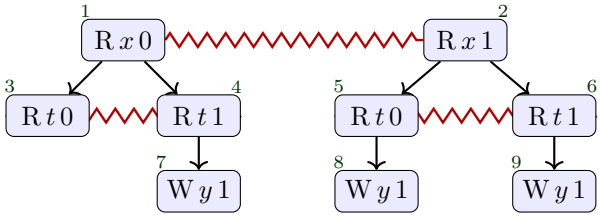
$$\mathsf{sdep} \coloneqq \bigcup_{i \in \{1, \cdots n\}} \mathit{freeze}(C_i \vdash w_i)$$

X_1 must be a *coherent execution*. We prove that for a coherent execution there always exists a choice of write justifications that freeze into dependencies to form a coherent execution.

We will illustrate freezing of the program,

$$\texttt{r}_1 \texttt{ := x; } \texttt{r}_2 \texttt{ := t; if(}\texttt{r}_1 == 1 \vee \texttt{r}_2 == 1\texttt{)\{y := 1\}}$$

whose event structure is as follows:

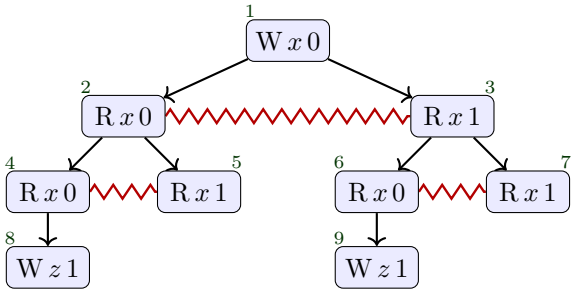


The rules later on in this section will provide us with justifications $\{(6 : \mathrm{R}\, t\, 1)\} \vdash (9 : \mathrm{W}\, y\, 1)$ and $\{(2 : \mathrm{R}\, x\, 1)\} \vdash (9 : \mathrm{W}\, y\, 1)$ (but not the *independent justification* $\vdash (9 : \mathrm{W}\, y\, 1)$). So in this program there are two *minimal* justifications of $(9 : \mathrm{W}\, y\, 1)$. The result of freezing is to duplicate all partial executions for each choice of write justifications. In this case, we get an execution containing $2 \xrightarrow{\mathsf{sdep}} 9$ and another one containing $6 \xrightarrow{\mathsf{sdep}} 9$.

5.4.1 Prefixing single events

When prefixing loads and stores, we model forwarding optimisations by updating the justification relation: e.g. when prefixing a write, $(\mathrm{w} : \mathrm{W}\, \mathrm{x}\, 0)$, to an event structure where $\{(r : \mathrm{R}\, x\, 0)\} \vdash w'$,

write forwarding satisfies the read of the justification, leaving an independently justified write, $\vdash w'$.



Forwarding is forbidden if there exists e in E such that $w \leq e \leq r$, as in the example above. In this example we do not forward 1 to 6. The rules of this section give us that $\{1, 3, 6\} \vdash 9$: we have preserved program order over the accesses of x, $1 \leq 3 \leq 6$, and we do not forward across the intervening read 3.

Read Semantics

We now define the semantics of read prefixing as follows:

$$(r : \mathrm{R}\, x\, v) \bullet (E_1, S_1, \vdash_1, \leq_1) \triangleq ((r : \mathrm{R}\, x\, v) \bullet E_1, S, \vdash, \leq) \tag{21}$$

where preserved program order $\leq$ is built straightforwardly out of $\leq_1$, ordering locks, unlocks and same-location accesses, and S is defined as the set of all $(X \cup \{r\}, \mathsf{lk}_X, \mathsf{rf}_X, \mathsf{sdep}_X)$, where X is a partial execution of S_1 and $\vdash$ is the smallest relation such that for all $C \vdash_1 e$ we have

$$C_1 \cup \{r\} \setminus \mathrm{LF} \vdash e$$

with LF being the "*Load Forwarded*" set of reads, *i.e.* the set of reads consecutively following the matching prefixed one:

$$\mathrm{LF} = \{(r' : \mathrm{R}\, x\, v) \in C_1 \mid \nexists e', r \leq^{\mathcal{X}} e' \leq^{\mathcal{X}} r'\}$$

This allows for load forwarding optimisations and coherence is satisfied by construction.

Write Semantics

The write semantics are then defined as follows:

$$(w : \mathrm{W}\, x\, v) \bullet (E_1, S_1, \vdash_1, \leq_1) \triangleq ((w : \mathrm{W}\, x\, v) \bullet E_1, S, \vdash, \leq) \tag{22}$$

where $\leq$ is built as in the read rule and S contains all *coherent* executions of the form,

$$(X \cup \{w\}, \mathsf{lk}_X, (\mathsf{rf}_X \cup \mathsf{rf}_i), \mathsf{sdep}_X)$$

where $X \in S_1$, and $w \xrightarrow{\mathsf{rf}_i} r$ for any set of matching reads r in E_1 such that condition (2.2) of coherence is satisfied. Adding rf_i edges leaves condition (2.1) satisfied.

The justification relation $\vdash$ is the smallest upward-closed relation such that for all $C \vdash_1 e$:

1. $\vdash w$
2. $C \setminus \mathrm{SF} \cup \{w\} \vdash e$ if there exists $e' \in C$ s.t. $w \leq^{\mathcal{X}} e'$
3. $C \setminus \mathrm{SF} \vdash e$ otherwise

with SF being the *Store Forwarding* set of reads, *i.e.* the set of reads that we are going to remove from the justification set for later events that are matching the write we are prefixing. This is defined as follows:

$$\mathrm{SF} = \{(r' : \mathrm{R}\, x\, v) \mid \nexists e, w \leq^{\mathcal{X}} e \leq^{\mathcal{X}} r'\}$$

When prefixing a write to an event structure, we add it to justifications that contain a read to the same variable. Failing to do so would invalidate the DRF-SC property.

5.4.2 Coproduct semantics

The coproduct mechanism is responsible for making writes independent of prior reads if they are sure to happen, regardless of the value read. It produces the independent writes that enabled relaxed behaviour in the examples in §5.1.

In the definition of coproduct we use an upward-closure of justification to enable the lifting of more dependencies. Whenever $C \vdash e$ we define $\uparrow(C)$ as the upward-closed justification set, i.e. $D \vdash e$ if $C \vdash e$, D is a conflict-free lock-free set with $C \subseteq D$, such that for all $e' \in D$ if e'' is an event such that $e'' \leq e'$ then $e'' \in D$.

Now we define the coproduct operation. If E_1 is a labelled event structure of the form $(r_1 : \mathrm{R}\, x\, v_1) \bullet E_1'$ and, similarly, E_2 is of the form $(r_2 : \mathrm{R}\, x\, v_2) \bullet E_2'$, the coproduct of event

structures is defined as,

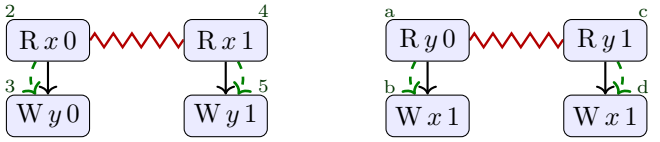
$$(E_1, S_1, \vdash_1, \leq_1) + (E_2, S_2, \vdash_2, \leq_2) \triangleq (E_1 + E_2, S_1 \cup S_2, (\vdash_1 \cup \vdash_2 \cup \vdash), \leq)$$

where whenever $\{r_1\} \cup C_1 \vdash_1 (w : \mathrm{W}\, y\, v)$ and $\{r_2\} \cup C_2 \vdash_2 (w' : \mathrm{W}\, y\, v)$ then if the following conditions hold, we have $D' \vdash w$ and $D'' \vdash w'$:

1. there exists a $D' \in \uparrow (C_1)$ that is isomorphic to a $D'' \in \uparrow (C_2)$, that is, there exists $f : D' \to D''$ that is a λ-preserving and $\leq^{\mathcal{X}}$-preserving bijection,
2. there is no event e in D' such that $r_1 \leq^{\mathcal{X}} e$

The example of §5.1 illustrates the application of condition (1) of coproduct. We begin by recalling LB_1 and LB_3, and their corresponding event structures below.

$(\mathtt{r_1 := x;\ y := r_1})$ (LB_1) $(\mathtt{r_2 := y;\ x := 1})$ (LB_3)



In each case, the event structure is built as the coproduct of the conflicting events. In (LB_3), prior to applying coproduct we have $\{a\} \vdash b$ and $\{c\} \vdash d$. The writes have the same label for both read values so, taking C_1 and C_2 to be empty, coproduct makes them independent, adding the independent writes $\vdash b$ and $\vdash d$. In contrast, the values of writes 3 and 5 differ in (LB_1), so the coproduct has $\{2\} \vdash 3$ and $\{4\} \vdash 5$. When ultimately frozen, the justifications of (LB_1) will produce the dependency edges $(2, 3)$ and $(4, 5)$ as described in Section 5.1.

As for condition (2), if there is an event in the justification set that is ordered in $\leq^{\mathcal{X}}$ with the respective top read, then the top read cannot be erased from the justification. Doing so would break the $\leq^{\mathcal{X}}$ link.

When having value sets that contain more than two values, we use $\sum_{v \in \mathcal{V}}$ to denote a *simultaneous coproduct* (rather than the infinite sum). More precisely, if we coproduct the event structures $E_0, E_1, \cdots, E_n$ in a pairwise fashion as follows,

$$(\cdots (E_0 + E_1) + \cdots) + E_v$$

we would get liftings that are undesirable. To see this, it suffices to consider the program,

$$\texttt{if(r==3)\{x := 2\}\{x := 1\}}$$

where the write to x of 1 is independent for a coproduct over values 1 and 2, but not when considering the event structure following $(\mathrm{R}\, x\, 3)$.

5.4.3 Lock semantics

When prefixing a lock, we order the lock before following events in $\leq$ and we freeze the justifications into dependencies. By freezing, we prevent justifications from events after the lock from interacting with newly appended events. This disables optimisations across the lock, e.g. store and load forwarding.

We define the semantics of locks as follows,

$$(l : \mathsf{L}) \bullet (E_1, \vdash_1, S_1, \leq_1) \triangleq ((l : \mathsf{L}) \bullet E_1, \emptyset, S, \leq) \tag{23}$$

where $\leq^{\mathcal{X}}$ remains unchanged and $(E_1', \emptyset, S_1', \leq_1') = \mathit{freeze}(E_1, \vdash_1, S_1, \leq_1)$, where S contains all partial executions of the form,

$$(X \cup \{l\}, (\mathsf{lk}_X \cup \mathsf{lk}), \mathsf{sdep}_X, \mathsf{rf}_X)$$

where $X \in S_1'$ and the lock order lk is such that for all lock or unlock event $l' \in X$, $l \xrightarrow{\mathsf{lk}} l'$. Finally, $\leq^{\mathsf{L}}$ is $\leq^{\mathsf{L}}{}_1'$ extended with the lock ordered before all events in E_1'.

The semantics for the unlock is similar.

5.4.4 Parallel composition

We define the parallel semantics as follows. Note that this operation freezes the constituent denotations before combining them, erasing their respective justification relations. This choice prevents the optimisation of dependencies across forks and it makes thread inlining optimisations unsound, as they are in the Promising Semantics (Kang et al. 2017) and the Java memory model (Manson, Pugh and Adve 2005).

$$(E_1, S_1, \vdash_1, \leq_1) \times (E_2, S_2, \vdash_2, \leq_2) \triangleq (E_1 \times E_2, S, \emptyset, \leq_1 \cup \leq_2)$$

where, S are all *coherent* partial executions of the form,

$$(X_1 \cup X_2, (\mathsf{lk}_{X_1} \cup \mathsf{lk}_{X_2} \cup \mathsf{lk}), (\mathsf{sdep}_{X_1} \cup \mathsf{sdep}_{X_2}), (\mathsf{rf}_{X_1} \cup \mathsf{rf}_{X_2} \cup \mathsf{rf}_e))$$

where $X_1 \in S_1^F$, $X_2 \in S_2^F$ and

- $\mathit{freeze}(E_1, S_1, \vdash_1, \leq_1) = (E_1, S_1^F, \emptyset, \leq_1)$
- $\mathit{freeze}(E_2, S_2, \vdash_2, \leq_2) = (E_2, S_2^F, \emptyset, \leq_2)$

Furthermore, lk is constrained so that $(\mathsf{lk}_{X_1} \cup \mathsf{lk}_{X_2} \cup \mathsf{lk})$ is a *total* order over the lock/unlock operations such that no lock/unlock operation is introduced between a lock and the next unlock on the same thread. Finally, we add all $(w : \mathrm{W}\, x\, v) \xrightarrow{\mathsf{rf}_e} (r : \mathrm{R}\, x\, v)$ edges such that the execution satisfies condition (2.1) of coherence[1] and such that w belongs to S_1^F and r belongs to S_2^F or vice versa.

5.5 Global memory model

5.5.1 Language

To abstract away the detail of C and C++'s syntax we take a simple language with the same expressiveness for writing shared-memory concurrent programs. This simplified syntax removes the concerns of type checking, objects, memory allocation, etc. which would otherwise be a distraction in this setting.

$$B := M = M \mid B \wedge B \mid B \vee B \mid \neg B$$

$$M := n \mid \texttt{r}$$

$$T ::= \texttt{skip} \mid \texttt{r := x} \mid \texttt{x := } M \mid T_1\texttt{;}\ T_2$$

$$\mid \textbf{if}(B)\{P_1\}\{P_2\} \mid \textbf{while}(B)\{P\} \mid \texttt{lock} \mid \texttt{unlock} \qquad P ::= T_1 \parallel \cdots \parallel T_n$$

5.5.2 Interpretation

With the syntax and the rules for composing event structures defined we can finally build an interpretation from syntax to event structure. The semantics is defined in a continuation passing style, meaning a program is evaluated 'bottom-up'. The continuation passing style is embodied

[1] Note that condition (2.2) does not need to be checked.

in the load and store interpretations, which evaluate the continuation first and then prefix an appropriate event, and the sequencing rule which builds a new continuation by creating recursive functions to the interpretation. The definitions are collected in Figure 45, but the interesting rules are explained below.

Load and Store. The store rule evaluates the continuation κ under the environment ρ to generate some event structure, to which a new write event is prefixed using the $\bullet$ operator.

$$[\![\texttt{r := x}]\!]_{n\ \rho\ \kappa} = \Sigma_{v \in V}(\mathrm{R}\, x\, v \bullet \kappa(\rho[r \mapsto v]))$$

The load rule is a little more complex. To interpret a load of `x` into an register `r`, every each value v in the value set $\mathcal{V}$ generates a new event structure, which are composed using the coproduct rule. Each of those event structures is built by taking the continuation κ evaluated with an updated environment ρ where `r` gets the value v.

$$[\![\texttt{x := } M]\!]_{n\ \rho\ \kappa} = (\mathrm{W}\, x\, [\![M]\!]_\rho) \bullet \kappa(\rho)$$

Sequencing. The composition of two blocks of program text is done by creating a continuation which evaluates the semantics of P_2 under the interpretation function, and passes that as a parameter to the interpretation of P_1. As each of the rules evaluate the continuation first, it is clear to see why this rule makes the interpretation 'bottom-up'.

$$[\![P_1\texttt{;}\ P_2]\!]_{n\ \rho\ \kappa} = [\![P_1]\!]_{n\ \rho\ (\lambda\rho.[\![P_2]\!]_{n\ \rho\ \kappa})}$$

Step Index. The interpretation is also step-indexed, which allows us to unroll while-loops for a finite number of steps – a limitation, but one which is not important for litmus test evaluation. This is manifest in two rules, the base case for the interpretation which yields an empty event structure when the step index becomes 1, and the while case which decrements the step index for each iteration of unrolling in the true branch.

$$[\![P]\!]_{1\ \rho\ \kappa} = \underline{\emptyset}$$

$$[\![\textbf{while}(B)\{P\}]\!]_{n\ \rho\ \kappa} = \begin{cases} [\![P\texttt{;}\ \textbf{while}(B)\{P\}]\!]_{(n-1)\ \rho\ \kappa} & [\![B]\!]_\rho = \text{true} \\ [\![\texttt{skip}]\!]_{n\ \rho\ \kappa} & [\![B]\!]_\rho = \text{false} \end{cases}$$

Parallel Composition. Parallel composition is carefully defined to induce a freeze at the thread boundaries. It puts a `lock` and `unlock` pair before and after the parallel program in order to ensure that any preceding code does not remove dependencies through the coproduct mechanism. The `lock`/`unlock` pair causes a freeze which moves justifications into sdep, and as coproduct only affects justification edges – not sdep edges – we can be sure that any dependencies internal to threads will not be elided by a preceding block of code.

$$
\begin{aligned}
[\![P_1 \parallel P_2]\!]_{n\ \rho\ \kappa} &= [\![\texttt{lock;unlock}]\!]_{n\ \rho\ \kappa'} \\
&\quad \text{where} \\
&\qquad \kappa' = (\lambda\rho.([\![P_1]\!]_{n\ \rho\ \emptyset}) \times ([\![P_2]\!]_{n\ \rho\ \emptyset}) \star ([\![\texttt{lock; unlock}]\!]_{n\ \rho\ \kappa}))
\end{aligned}
$$

$$
\begin{aligned}
[\![P]\!]_{1\ \rho\ \kappa} &= \underline{\emptyset} \\
[\![\texttt{skip}]\!]_{n\ \rho\ \kappa} &= \kappa(\rho, \psi) \\
[\![\texttt{r := x}]\!]_{n\ \rho\ \kappa} &= \Sigma_{v \in V}(\mathrm{R}\, x\, v \bullet \kappa(\rho[r \mapsto v])) \\
[\![\texttt{x := } M]\!]_{n\ \rho\ \kappa} &= (\mathrm{W}\, x\, [\![M]\!]_\rho) \bullet \kappa(\rho) \\
[\![P_1\texttt{;}\ P_2]\!]_{n\ \rho\ \kappa} &= [\![P_1]\!]_{n\ \rho\ (\lambda\rho.[\![P_2]\!]_{n\ \rho\ \kappa})} \\
[\![P_1 \parallel P_2]\!]_{n\ \rho\ \kappa} &= [\![\texttt{lock;unlock}]\!]_{n\ \rho\ \kappa'} \\
&\quad \text{where} \\
&\qquad \kappa' = (\lambda\rho.([\![P_1]\!]_{n\ \rho\ \emptyset}) \times ([\![P_2]\!]_{n\ \rho\ \emptyset}) \star ([\![\texttt{lock; unlock}]\!]_{n\ \rho\ \kappa})) \\
[\![\textbf{if}(B)\{P_1\}\{P_2\}]\!]_{n\ \rho\ \kappa} &= \begin{cases} [\![P_1]\!]_{n\ \rho\ \kappa} & [\![B]\!]_\rho = \text{true} \\ [\![P_2]\!]_{n\ \rho\ \kappa} & [\![B]\!]_\rho = \text{false} \end{cases} \\
[\![\textbf{while}(B)\{P\}]\!]_{n\ \rho\ \kappa} &= \begin{cases} [\![P\,\textbf{;while}(B)\{P\}]\!]_{(n-1)\ \rho\ \kappa} & [\![B]\!]_\rho = \text{true} \\ [\![\texttt{skip}]\!]_{n\ \rho\ \kappa} & [\![B]\!]_\rho = \text{false} \end{cases} \\
[\![\texttt{lock}]\!]_{n\ \rho\ \kappa} &= (\mathsf{L} \bullet E_1, \vdash_1) \\
&\quad \text{where } (E_1, \vdash_1) = \kappa(\rho) \\
[\![\texttt{unlock}]\!]_{n\ \rho\ \kappa} &= (\mathsf{U} \bullet E_1, \vdash_1) \\
&\quad \text{where } (E_1, \vdash_1) = \kappa(\rho)
\end{aligned}
$$

Figure 45: Collected definition for the interpretation function from program syntax to event structure.

5.6 Conclusion

In this chapter we have developed a foundation for defining a memory model with a semantic dependency relation. Program interpretation builds the meaning of dependency stepwise, with each composition unlocking new ways that a dependency can be elided. Our coproduct mechanism is at the heart of this, recognising symmetries between different future executions of the program,

by comparing the justification relations of a program for each choice of read being prefixed. Eventually, this leaves us with a sdep that can be exported on a per-execution basis and be combined with the axioms of traditional axiomatic memory models to provide a full-fat thin-air free memory model. This idea will be explored in the next chapter as we expand this model to a full C++ semantics.

Chapter 6

Fixing C11

To fix the definition of C/C++11 we need a thin-air free semantics which preserves the semantics of synchronisation. With MRD providing a definition of sdep it is easy to incorporate existing axiomatic memory models to filter the executions of MRD. We call these new models MRD-augmented models, and we explore two such models in this chapter. MRD+IMM is an augmented IMM model which shows that a model with sdep can be efficiently compiled to hardware. MRD+RC11 is a full-fat C/C++ memory model, supporting fences, release/acquire memory accesses, and atomic read-modify-write operations. To validate that our incorporation of MRD is a strict extension to allow load buffering behaviours, we prove that any execution allowed by a non-augmented model is also allowed by the MRD-augmented version. In particular, the MRD+RC11 proof validates that the MRD-augmented model of RC11 is a suitable C/C++ memory model candidate, as it forbids no executions previously allowed by RC11 and strictly allows new optimisations. To contextualise this model §6.6 compares the MRD-augmented models to other thin-air free programming language models.

6.1 Intermediate Memory Model (IMM) in Event structures

We use the models specified by Podkopaev, Lahav and Vafeiadis (2019) for both IMM and RC11, this allows us to avoid replicated effort in the definitions of the relations and axioms common to both models.

$$
\begin{aligned}
o_{\mathrm{R}} &::= \mathsf{rlx} \mid \mathsf{acq} \\
o_{\mathrm{W}} &::= \mathsf{rlx} \mid \mathsf{rel} \\
o_{\mathrm{F}} &::= \mathsf{acq} \mid \mathsf{rel} \mid \mathsf{acq\_rel} \mid \mathit{sc} \\
o_{\mathrm{RMW}} &::= \mathit{normal} \mid \mathit{strong}
\end{aligned}
$$

$$
\begin{aligned}
&B := M = M \mid B \wedge B \mid B \vee B \mid \neg B \\
&M := n \mid \mathtt{r} \\
&T ::= \mathtt{skip} \mid \mathtt{r} :=^{o_{\mathrm{R}}} \mathtt{x} \mid \mathtt{x} :=^{o_{\mathrm{W}}} M \mid T_1 \,;\, T_2 \\
&\qquad \mid \mathbf{if}(B)\{P_1\}\{P_2\} \mid \mathbf{while}(B)\{P\} \mid \mathtt{lock} \mid \mathtt{unlock} \\
&\qquad \mid \mathtt{r} := \mathbf{FADD}^{o_{\mathrm{R}},o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}(\mathtt{x}, M) \mid \mathbf{CAS}^{o_{\mathrm{R}},o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}(\mathtt{x}, M, M) \mid \mathbf{fence}^{o_{\mathrm{F}}} \\
&P ::= T_1 \parallel \cdots \parallel T_n
\end{aligned}
$$

Figure 46: The Language considered by the IMM memory model.

6.1.1 Pre-execution semantic model

First we provide a model, written (for a program P) as $[\![P]\!]_{\mathsf{MRD+IMM}}$, that combines our relaxed dependencies with the axiomatic model of IMM, here written as $[\![P]\!]_{\mathsf{IMM}}$. We will make these definitions precise shortly. We then show that $[\![P]\!]_{\mathsf{MRD+IMM}}$ is weaker than $[\![P]\!]_{\mathsf{IMM}}$, establishing that $[\![P]\!]_{\mathsf{MRD+IMM}}$ is implementable over hardware architectures like x86-TSO, ARMv7, ARMv8 and Power. Secondly, we relax the RC11 axiomatic model by using our relaxed dependencies model MRD to create a new model $[\![P]\!]_{\mathsf{MRD\text{-}C11}}$, and show this model weaker than the RC11 model. We argue that the mathematical description of $[\![P]\!]_{\mathsf{MRD\text{-}C11}}$ is lightweight and close to the C++ standard, it would therefore require minimal work to augment the standard with the ideas presented in this chapter.

To prove implementability over hardware architectures we define a *pre-execution* semantics, where the relaxed dependency relation sdep is calculated along with the data and control dependencies from IMM. To combine our model with IMM, we redefine the ar relation such that it is parametrised by an arbitrary relation which we put in place of the relations (data ∪ ctrl), *i.e.* ar(data ∪ ctrl) is the original relation, and ar(sdep) is the same relation with sdep put in place of data ∪ ctrl. We define the executions in $[\![P]\!]_{\mathsf{MRD+IMM}}$ as the maximal conflict-free sets such that ar(sdep) is acyclic, and executions in $[\![P]\!]_{\mathsf{IMM}}$ as the maximal conflict-free sets such that ar(data ∪ ctrl) is acyclic. These will be defined formally in §6 and §5, respectively.

We then define $[\![P]\!]_{n\ \iota\ \kappa}$ such that it returns an event structure augmented with a record rels of relaxed dependency relations: sdep and the IMM dependencies data (data dependencies), ctrl (control dependencies) and casdep (CAS dependencies). The augmented pre-semantics, namely

$\langle P \rangle^{\dagger}_{n\ \iota\ \kappa}$, calculates sdep by first replacing preserved program order $\leq$ with the relation par (the preserved order relation of IMM, see 6.2):

$$\mathsf{par} \triangleq \mathsf{bob} \cup ([R]; \mathsf{deps}(\emptyset); [W]) \cup ([W_{strong}]; \sqsubseteq; [W])$$

with dependencies then calculated as usual. When interpreting read r, for example, instead of gathering the matching reads which follow r in $\leq$ we look for reads which follow r in par. Note that par is the part of ar from IMM that is statically generated and avoids using data and ctrl – deps($\emptyset$) captures this change to the IMM dependency relation. This small change in the calculation of the dependencies is necessary to prove $[\![P]\!]_{\mathsf{MRD+IMM}}$ is weaker than $[\![P]\!]_{\mathsf{IMM}}$, as we show shortly.

After the pre-semantics is calculated, we filter out the executions using the IMM axioms. In particular, we define $[\![P]\!]_{\mathsf{IMM}}$ as the set of maximal conflict-free sets that satisfy the IMM axioms (see Definition 5 for the full list). Then, we define $[\![P]\!]_{\mathsf{MRD+IMM}}$ as the set of maximal conflict-free sets that satisfy a variant of the IMM axioms where (data $\cup$ ctrl) is replaced with sdep, for sdep $\in S$.

The signature Σ of labels is defined as follows

- Read label : $\mathsf{R}^{o_{\mathrm{R}}}_{s}\ x\ v$ where $s \in \{\text{not-ex}, \text{ex}\}$
- Write label : $\mathsf{W}^{o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}\ x\ v$
- Fence label : $\mathrm{F}^{o_{\mathrm{F}}}$
- Lock labels : L and U

This is an augmentation of the previous definition of labels – see Section 1 – these extra annotations are needed to record exclusivity mode information required for implementing read-modify-write operations on ARM under the IMM model.

We now define a pre-semantics as a function into an event structure with additional relations. First we define an environment ι to be a record of environments (ρ, ψ, γ) where $\rho : \mathcal{R} \to \mathcal{V}$ is a register environment taking registers to their values, $\psi : \mathcal{R} \to \mathcal{P}(E.R)$ maps registers to the set of events that were used to compute the register's value, and $\gamma : \mathcal{P}(E.R)$ is a set of events which have influenced the control flow up to this point in the continuation.

Our domain will be an event structure $(E, \sqsubseteq, \#)$, E for brevity, along with a record of relations rels made out of the relations (data, ctrl, casdep, addr) whose types are defined as follows. For an event structure E,

- $\mathsf{data}_E \subseteq [E.R]; \sqsubseteq; [E.W]$, called *data dependency*
- $\mathsf{addr}_E \subseteq [E.R]; \sqsubseteq; [E.R \cup E.W]$, called *address dependency*
- $\mathsf{ctrl}_E \subseteq [E.R]; \sqsubseteq; [E.R_{\text{ex}}]$, called *control dependency*, that is forward-closed under program order: $\mathsf{ctrl}_E; \sqsubseteq \subseteq \mathsf{ctrl}_E$
- $\mathsf{casdep}_E \subseteq [E.R]; \sqsubseteq; [E.R_{\text{ex}}]$, called *CAS dependency*

We now define the pre-semantic function. For a program P,a step-index n for the while-loops, a record of environments ι and a continuation κ, the function $\langle P \rangle^{\dagger}_{n\ \iota\ \kappa}$ returns an event structure (E, rels) and defined by first interpreting the top-level parallel threads as

$$\langle T_1 \parallel \cdots \parallel T_m \rangle^{\dagger}_{n} = \mathit{freeze}\left(\langle \texttt{Init} \rangle^{\dagger}_{n\ \emptyset\ \lambda\iota.\langle \texttt{lock; unlock} \rangle^{\dagger}_{n\ \iota\ \lambda\iota.\left(\langle T_1 \rangle^{\dagger}_{n\ \iota\ \emptyset} \times \cdots \times \langle T_m \rangle^{\dagger}_{n\ \iota\ \emptyset}\right)}} \right)$$

where `Init` is a sequence of writes to zero for all locations used by the threads $T_1, \cdots, T_n$. Then each thread is interpreted by induction on the step-index n and then by induction on T. The rules are introduced in turn with an explanation for the differences compared to the rules of §5.5, and the definitions are collected in Figure 47.

Language constructions with minimal updates. The simple base cases for this semantics are where we reach a `skip` command, or where the step-index n reaches 1. For **fence** statements, the same prefixing happens except the continuation is evaluated with an ι instead of simply a ρ. Similarly, the sequence rule is the same as before, except that now the continuation is constructed to take an ι.

$$\begin{aligned}
\langle P \rangle^{\dagger}_{1\ \iota\ \kappa} &= \underline{\emptyset} \\
\langle \texttt{skip} \rangle^{\dagger}_{n\ \iota\ \kappa} &= \kappa(\iota) \\
\langle \mathbf{fence}^{o_{\mathrm{F}}} \rangle^{\dagger}_{n\ \iota\ \kappa} &= (\mathrm{F}^{o_{\mathrm{F}}} \bullet E_\kappa, \mathsf{rels}_\kappa) \\
&\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota) \\
\langle T_1 \,;\, T_2 \rangle^{\dagger}_{n\ \iota\ \kappa} &= \langle T_1 \rangle^{\dagger}_{n\ \iota\ (\lambda\iota'.\langle T_2 \rangle^{\dagger}_{n\ \iota'\ \kappa})}
\end{aligned}$$

Load. In the load case, some additional information is stored compared to the semantics presented in the previous chapter. Here, $\iota.\rho$ is updated as before – recording the value the register stores in a local environment, but also $\iota.\psi$ is updated to record that a named register gets its

value from a particular event. This ψ environment will be used to create data dependencies (data).

$$\langle \texttt{r} :=^{o_R} \texttt{x} \rangle^{\dagger}_{n\ \iota\ \kappa} = (\Sigma_{v \in V}((r_v : \mathsf{R}^{o_R}_{\text{not-ex}}\ x\ v) \bullet E_\kappa), \mathsf{rels}_\kappa)$$
$$\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota')$$
$$\iota'.\rho = \iota.\rho[r \mapsto v]$$
$$\iota'.\psi = \iota.\psi[r \mapsto \{r_v\}]$$

Store. The store case reveals the first use of $\iota.\psi$ and $\iota.\gamma$. Here, for each register in the expression M a set is project from ψ and γ. The set from ψ records which events generated the value stored in $\texttt{r}$, so is turned into a new data dependency edge in rels. The set from γ is all the events which created control syntactic dependencies to this point in the event structure, and is used to create new ctrl edges in rels.

$$\langle \texttt{x} :=^{o_W} M \rangle^{\dagger}_{n\ \iota\ \kappa} = ((w : \mathsf{W}^{o_W}_{normal}\ x\ [\![M]\!]_{\iota.\rho}) \bullet E_k, \mathsf{rels})$$
$$\text{where } (E_k, \mathsf{rels}_\kappa) = \kappa(\iota)$$
$$\mathsf{rels.data} = \mathsf{rels}_\kappa\mathsf{.data} \cup \bigcup_{r \in M} (\psi(r) \times w)$$
$$\mathsf{rels.ctrl} = \mathsf{rels}_\kappa\mathsf{.ctrl} \cup \bigcup_{r \in \iota.\gamma} \{(r, w)\}$$

Control flow. The **while** and **if** constructions introduce new control flow. As such for each register in the boolean test B, the events which gave the registers those values are looked up in the map $\iota.\psi$ and added to the set of events which generate the control flow – $\iota.\gamma$ – and passed into the respective continuations.

$$\langle \textbf{while}(B)\{T\} \rangle^{\dagger}_{n\ \iota\ \kappa} = \begin{cases} \langle T ; \textbf{while}(B)\{T\} \rangle^{\dagger}_{(n-1)\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{true} \\ \langle \texttt{skip} \rangle^{\dagger}_{n\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{false} \end{cases}$$
$$\text{where } \iota'.\gamma = \iota.\gamma \cup \bigcup_{\texttt{r} \in B} \psi(\texttt{r})$$

$$\langle \mathbf{if}(B)\{T_1\}\{T_2\}\rangle^{\dagger}_{n\ \iota\ \kappa} = \begin{cases} \langle T_1\rangle^{\dagger}_{n\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{true} \\ \langle T_2\rangle^{\dagger}_{n\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{false} \end{cases}$$
$$\text{where } \iota'.\gamma = \iota.\gamma \cup \bigcup_{\mathtt{r}\in B} \psi(\mathtt{r})$$

Atomic Copy and Store (CAS). For CAS statements the $\iota.\phi$ and $\iota.\rho$ environments are updated just as they are in the load rule, but here there is some further house keeping to do. A new rmw edge is recorded from the event corresponding to the successful read event (r_v^{success}), to the write event (w), and a casdep edge is created between the write of the successful case and every read event required to construct the value found in $\mathtt{r}_1$ according to $\iota.\psi$.

$$\langle \mathtt{r}_1 := \mathbf{CAS}^{o_{\mathrm{R}},o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}(\mathtt{x}, M_1, M_2)\rangle^{\dagger}_{n\ \iota\ \kappa} =$$
$$\sum_{v\in\mathcal{V}} \begin{cases} ((r_v^{\text{success}} : \mathsf{R}^{o_{\mathrm{R}}}_{\mathrm{ex}}\ x\ v) \bullet (w : \mathsf{W}^{o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}\ x\ [\![M]\!]_{\iota.\rho[\mathtt{r}_1\mapsto v]}) \bullet E_\kappa, \mathsf{rels}) & \text{if } [\![M_1]\!]_{\iota.\rho} = v \\ ((r_v^{\text{failure}} : \mathsf{R}^{o_{\mathrm{R}}}_{\mathrm{ex}}\ x\ v) \bullet E_\kappa, \mathsf{rels}) & \text{otherwise} \end{cases}$$
$$\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota')$$
$$\iota'.\psi = \iota.\psi[\mathtt{r}_1 \mapsto \{r_v\}]$$
$$\iota'.\rho = \iota.\rho[\mathtt{r}_1 \mapsto v]$$
$$\mathsf{rels}.\mathsf{rmw} = \{(r_v^{\text{success}}, w)\} \cup \mathsf{rels}_\kappa.\mathsf{rmw}$$
$$\mathsf{rels}.\mathsf{casdep} = \{w\} \times (\bigcup_{\mathtt{r}\in M_1} \psi(\mathtt{r})) \cup \mathsf{rels}_\kappa.\mathsf{casdep}$$

Atomic Fetch and Add (FADD). Similarly to CAS, the Fetch and Add rule updates the environments $\iota.\psi$ and $\iota.\rho$, but there are some differences in the recorded relations. Here, a rmw edge is recorded between all reads and their subsequent writes, and no casdep edges are recorded.

$$\langle \mathtt{r}_1 := \mathbf{FADD}^{o_{\mathrm{R}},o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}(\mathtt{x}, M)\rangle^{\dagger}_{n\ \iota\ \kappa} = (\sum_{v\in\mathcal{V}} (r_v : \mathsf{R}^{o_{\mathrm{R}}}_{\mathrm{ex}}\ x\ v) \bullet (w_v : \mathsf{W}^{o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}\ x\ [\![v + M]\!]_{\iota'.\rho}) \bullet E_\kappa, \mathsf{rels}_\kappa)$$
$$\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota')$$
$$\iota'.\psi = \iota.\psi[\mathtt{r}_1 \mapsto \{r_v\}]$$
$$\iota'.\rho = \iota.\rho[\mathtt{r}_1 \mapsto v]$$
$$\mathsf{rels}.\mathsf{rmw} = \bigcup_{v\in V} (r_v, w_v) \cup \mathsf{rels}_\kappa.\mathsf{rmw}$$

Definition 3 (Intra-thread and extra-thread relations)**.** *We define a pair of relation restriction operator which give relations edges exclusively within one thread, and exclusively between threads*

$$
\begin{aligned}
\langle P\rangle^{\dagger}_{1\ \iota\ \kappa} &= \underline{\emptyset}\\
\langle \mathtt{skip}\rangle^{\dagger}_{n\ \iota\ \kappa} &= \kappa(\iota)\\
\langle \mathbf{fence}^{o_{\mathrm{F}}}\rangle^{\dagger}_{n\ \iota\ \kappa} &= (\mathrm{F}^{o_{\mathrm{F}}} \bullet E_\kappa, \mathsf{rels}_\kappa)\\
&\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota)\\
\langle T_1\,;\, T_2\rangle^{\dagger}_{n\ \iota\ \kappa} &= \langle T_1\rangle^{\dagger}_{n\ \iota\ (\lambda\iota'.\langle T_2\rangle^{\dagger}_{n\ \iota'\ \kappa})}\\
\langle \mathtt{r} :=^{o_{\mathrm{R}}} \mathtt{x}\rangle^{\dagger}_{n\ \iota\ \kappa} &= (\Sigma_{v\in V}((r_v : \mathsf{R}^{o_{\mathrm{R}}}_{\text{not-ex}}\ x\ v) \bullet E_\kappa), \mathsf{rels}_\kappa)\\
&\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota')\\
\iota'.\rho &= \iota.\rho[r \mapsto v]\\
\iota'.\psi &= \iota.\psi[r \mapsto \{r_v\}]\\
\langle \mathtt{x} :=^{o_{\mathrm{W}}} M\rangle^{\dagger}_{n\ \iota\ \kappa} &= ((w : \mathsf{W}^{o_{\mathrm{W}}}_{normal}\ x\ [\![M]\!]_{\iota.\rho}) \bullet E_k, \mathsf{rels})\\
&\text{where } (E_k, \mathsf{rels}_\kappa) = \kappa(\iota)\\
\mathsf{rels}.\mathsf{data} &= \mathsf{rels}_\kappa.\mathsf{data} \cup \bigcup_{r\in M} (\psi(r) \times w)\\
\mathsf{rels}.\mathsf{ctrl} &= \mathsf{rels}_\kappa.\mathsf{ctrl} \cup \bigcup_{r\in \iota.\gamma} \{(r, w)\}
\end{aligned}
$$

$$
\langle \mathbf{while}(B)\{T\}\rangle^{\dagger}_{n\ \iota\ \kappa} = \begin{cases} \langle T\,;\mathbf{while}(B)\{T\}\rangle^{\dagger}_{(n-1)\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{true}\\ \langle \mathtt{skip}\rangle^{\dagger}_{n\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{false}\end{cases}
$$

$$
\text{where } \iota'.\gamma = \iota.\gamma \cup \bigcup_{\mathtt{r}\in B} \psi(\mathtt{r})
$$

$$
\langle \mathbf{if}(B)\{T_1\}\{T_2\}\rangle^{\dagger}_{n\ \iota\ \kappa} = \begin{cases} \langle T_1\rangle^{\dagger}_{n\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{true}\\ \langle T_2\rangle^{\dagger}_{n\ \iota'\ \kappa} & [\![B]\!]_{\iota.\rho} = \text{false}\end{cases}
$$

$$
\text{where } \iota'.\gamma = \iota.\gamma \cup \bigcup_{\mathtt{r}\in B} \psi(\mathtt{r})
$$

$$
\begin{aligned}
&\langle \mathtt{r}_1 := \mathbf{CAS}^{o_{\mathrm{R}},o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}(\mathtt{x}, M_1, M_2)\rangle^{\dagger}_{n\ \iota\ \kappa} =\\
&\quad \sum_{v\in\mathcal{V}} \begin{cases} ((r_v^{\text{success}} : \mathsf{R}^{o_{\mathrm{R}}}_{\text{ex}}\ x\ v) \bullet (w : \mathsf{W}^{o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}\ x\ [\![M]\!]_{\iota.\rho[\mathtt{r}_1\mapsto v]}) \bullet E_\kappa, \mathsf{rels}) & \text{if } [\![M_1]\!]_{\iota.\rho} = v\\ ((r_v^{\text{failure}} : \mathsf{R}^{o_{\mathrm{R}}}_{\text{ex}}\ x\ v) \bullet E_\kappa, \mathsf{rels}) & \text{otherwise}\end{cases}\\
&\quad \text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota')\\
&\qquad \iota'.\psi = \iota.\psi[\mathtt{r}_1 \mapsto \{r_v\}]\\
&\qquad \iota'.\rho = \iota.\rho[\mathtt{r}_1 \mapsto v]\\
&\qquad \mathsf{rels}.\mathsf{rmw} = \{(r_v^{\text{success}}, w)\} \cup \mathsf{rels}_\kappa.\mathsf{rmw}\\
&\qquad \mathsf{rels}.\mathsf{casdep} = \{w\} \times (\bigcup_{\mathtt{r}\in M_1} \psi(\mathtt{r})) \cup \mathsf{rels}_\kappa.\mathsf{casdep}
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathtt{r}_1 := \mathbf{FADD}^{o_{\mathrm{R}},o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}(\mathtt{x}, M)\rangle^{\dagger}_{n\ \iota\ \kappa} &= (\sum_{v\in\mathcal{V}}(r_v : \mathsf{R}^{o_{\mathrm{R}}}_{\text{ex}}\ x\ v) \bullet (w_v : \mathsf{W}^{o_{\mathrm{W}}}_{o_{\mathrm{RMW}}}\ x\ [\![v + M]\!]_{\iota'.\rho}) \bullet E_\kappa, \mathsf{rels}_\kappa)\\
&\text{where } (E_\kappa, \mathsf{rels}_\kappa) = \kappa(\iota')\\
\iota'.\psi &= \iota.\psi[\mathtt{r}_1 \mapsto \{r_v\}]\\
\iota'.\rho &= \iota.\rho[\mathtt{r}_1 \mapsto v]\\
\mathsf{rels}.\mathsf{rmw} &= \bigcup_{v\in V} (r_v, w_v) \cup \mathsf{rels}_\kappa.\mathsf{rmw}
\end{aligned}
$$

Figure 47: Collected definitions for interpretation into MRD-augmented pre-semantics.

respectively.

$$R_i \triangleq R \cap (\sqsubseteq \cup \sqsubseteq^{-1}) \qquad \text{(Intra-Thread)}$$

$$R_e \triangleq R \setminus (\sqsubseteq \cup \sqsubseteq^{-1}) \qquad \text{(Extra-Thread)}$$

For example, rf *between events of the same thread would be written* rf_i*, and* co *between events of different threads would be written* co_e*.*

Definition 4 (Pre-execution)**.** *For a program* P*, let* (E, S, rels) *be* $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$*. A* pre-execution *for* P *consists of*

- *a* $\sqsubseteq$*-downward-closed conflict-free set* X *in* E
- *the relations in* rels_E
- *a relation* $\mathsf{rf}_X \subseteq X.W(x) \times X.R(x)$ *called* reads-from *for all* $x \in \mathcal{X}$ *and such that for all* $(w, r) \in \mathsf{rf}_X$ *we have* $(w : \mathrm{W}\, x\, v)$ *and* $(r : \mathrm{R}\, x\, v)$ *that is reads only read from a* matching write*. This relation is made of two parts*
  - $\mathsf{rf}_{e\,X}$ *when two events are on separate threads*
  - $\mathsf{rf}_{i\,X}$ *when two events are related in* $\sqsubseteq$
- *a strict partial order* $\mathsf{co}_X \subseteq X.W(x) \times X.W(x)$ *called* coherence order *or* modification order

6.2 IMM

Given an execution, we recall the definitions of Podkopaev, Lahav and Vafeiadis (2019) to get the following derived relations:

$$
\begin{aligned}
\mathsf{rs} &\triangleq [\mathsf{W}]; \sqsubseteq|_{loc}; [\mathsf{W}] \cup [\mathsf{W}]; (\sqsubseteq|^{?}_{loc}; \mathsf{rf}; \mathsf{rmw})^{*} \\
\mathsf{rel} &\triangleq ([\mathsf{W}^{\mathsf{rel}}] \cup [\mathsf{F}^{\sqsupseteq \mathsf{rel}}]; \sqsubseteq); \mathsf{rs} \\
\mathsf{sw} &\triangleq \mathsf{rel}; (\mathsf{rf}_i \cup \sqsubseteq|^{?}_{loc}; \mathsf{rf}_e); ([\mathsf{R}^{\mathsf{acq}}] \cup \sqsubseteq; [\mathsf{F}^{\sqsupseteq \mathsf{acq}}]) \\
\mathsf{hb} &\triangleq (\sqsubseteq \cup \mathsf{sw})^{+} \\
\mathsf{fr} &\triangleq \mathsf{rf}^{-1}; \mathsf{co} \\
\mathsf{eco} &\triangleq \mathsf{rf} \cup \mathsf{co}; \mathsf{rf}^{?} \cup \mathsf{fr}; \mathsf{rf}^{?} \\
\mathsf{deps}(dep) &\triangleq dep \cup \mathsf{addr}; \sqsubseteq^{?} \cup \mathsf{casdep} \cup [\mathsf{R}_{\mathrm{ex}}]; \sqsubseteq \\
\mathsf{ppo}(dep) &\triangleq [\mathsf{R}]; (\mathsf{deps}(dep) \cup \mathsf{rf}_i)^{+}; [\mathsf{W}] \\
\mathsf{detour} &\triangleq (\mathsf{co}_e; \mathsf{rf}_e) \cap \sqsubseteq \\
\mathsf{psc} &\triangleq [\mathsf{F}^{sc}]; \mathsf{hb}; \mathsf{eco}; \mathsf{hb}; [\mathsf{F}^{sc}] \\
\mathsf{bob} &\triangleq \sqsubseteq; [\mathsf{W}^{\mathsf{rel}}] \cup [\mathsf{R}^{\mathsf{acq}}]; \sqsubseteq \cup \sqsubseteq; [\mathsf{F}] \cup [\mathsf{F}]; \sqsubseteq \cup [\mathsf{W}^{\mathsf{rel}}]; \sqsubseteq|_{loc}; [\mathsf{W}] \\
\mathsf{ar}(dep) &\triangleq \mathsf{rf}_e \cup \mathsf{bob} \cup \mathsf{ppo}(dep) \cup \mathsf{detour} \cup \mathsf{psc} \cup [\mathsf{W}_{strong}]; \sqsubseteq; [\mathsf{W}]
\end{aligned}
$$

Finally, the set of executions that are accepted by this semantics is defined as follows.

Definition 5 (Set of executions in $[\![P]\!]_{\mathsf{IMM}}$)**.** *Let (E, S, rels) be $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$. An execution X belongs to $[\![P]\!]_{\mathsf{IMM}}$ iff X is a pre-execution of P – see Definition 4 – that satisfies the following properties:*

1. *if all the reads in X have a write that they read from*
2. *for every location $x \in \mathcal{X}$, co_X totally orders $X.W(x)$*
3. *$\mathsf{hb}_X; \mathsf{eco}^{=}_X$ is irreflexive*
4. *$\mathsf{rmw}_X \cap (\mathsf{fr}_X e; \mathsf{co}_X e) = \emptyset$*
5. *$\mathsf{ar}_X(\mathsf{data} \cup \mathsf{ctrl})$ acyclic*

6.3 MRD+IMM

Only a small change is needed to capture MRD+IMM, instead of using syntactic dependencies in the acyclicity check, we instead extract a choice of semantic dependency.

Definition 6 (Set of executions in $[\![P]\!]_{\mathsf{MRD+IMM}}$)**.** *Let* (E, S, rels) *be* $\langle P \rangle^{\dagger}_{n\,\emptyset\,\emptyset}$*. An execution* X *belongs to* $[\![P]\!]_{\mathsf{MRD+IMM}}$ *iff* X *is a pre-execution of* P *– see Definition 4 – that satisfies the following properties:*

1. *if all the reads in* X *have a write that they read from*
2. *for every location* $x \in \mathcal{X}$*,* co_X *totally orders* $X.W(x)$
3. $\mathsf{hb}_X ; \mathsf{eco}^{=}_X$ *is irreflexive*
4. $\mathsf{rmw}_X \cap (\mathsf{fr}_{eX} ; \mathsf{co}_{eX}) = \emptyset$
5. *for* $\mathsf{sdep} \in S_E$*,* $\mathsf{ar}_X(\mathsf{sdep})$ *acyclic*

Note that Condition 5 has been change with respect to Definition 5, and now includes sdep instead of (data ∪ ctrl). This modification is the intervention of MRD on IMM which forbids thin air executions.

6.4 RC11

We recall the additional derived relations of RC11 as defined by Podkopaev, Lahav and Vafeiadis (2019). By using the IMM-compatible expression of the RC11 relations, rather than the original definitions of Lahav et al. (2017), we can preserve the proof of efficient compilation mappings from Podkopaev, Lahav and Vafeiadis (2019).

Definition 7 (RC11 derived relations)**.**

$$\mathsf{rs}_{\mathsf{RC11}} \triangleq [W] ; \sqsubseteq|^{?}_{loc} ; (\mathsf{rf} ; \mathsf{rmw})$$
$$\mathsf{rel}_{\mathsf{RC11}} \triangleq ([W^{\mathsf{rel}}] \cup [F^{\sqsupseteq \mathsf{rel}}] ; \sqsubseteq) ; \mathsf{rs}_{\mathsf{RC11}}$$
$$\mathsf{sw}_{\mathsf{RC11}} \triangleq \mathsf{rel}_{\mathsf{RC11}} ; \mathsf{rf} ; ([R^{\mathsf{acq}}] \cup \sqsubseteq ; [F^{\sqsupseteq \mathsf{acq}}])$$
$$\mathsf{hb}_{\mathsf{RC11}} \triangleq (\sqsubseteq \cup \mathsf{sw}_{\mathsf{RC11}})^{+}$$

Definition 8 (RC11 executions)**.** *Let* (E, S, rels) *be* $\langle P \rangle^{\dagger}_{n\,\emptyset\,\emptyset}$*. An execution* X *belongs to* $[\![P]\!]_{RC11}$ *iff* X *is a pre-execution of* P *– see Definition 4 – that satisfies the following properties*

1. *if all the reads in* X *have a write that they read from*
2. *for every location* $x \in \mathcal{X}$*,* co_X *totally orders* $X.W(x)$
3. $\mathsf{hb}_{\mathsf{RC11}} ; \mathsf{eco}^{=}$ *is irreflexive*

4. $\mathsf{rmw} \cap (\mathsf{fr}_e; \mathsf{co}_e) = \emptyset$

5. $[F^{sc}]; (\mathsf{hb}_{\mathsf{RC11}} \cup \mathsf{hb}_{\mathsf{RC11}}; \mathsf{eco}; \mathsf{hb}_{\mathsf{RC11}}); [F^{sc}]$ *is acyclic*

6. $\sqsubseteq \cup \mathsf{rf}$ *is acyclic*

Taking all relations to be on the execution X*.*

6.5 MRD+RC11

With the definition of RC11 in place, we can define the augmented model of MRD+RC11. The approach is almost identical to that of MRD+IMM, weakening the strong acyclicity relation with semantic dependency (sdep).

Definition 9 (MRD +RC11 executions)**.** *Let* (E, S, rels) *be* $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$*. An execution* X *belongs to* $[\![P]\!]_{MRD+RC11}$ *iff* X *is a pre-execution of* P *– see Definition 4 – that satisfies the following properties*

1. *if all the reads in* X *have a write that they read from*

2. *for every location* $x \in \mathcal{X}$*,* co_X *totally orders* $X.W(x)$

3. $\mathsf{hb}_{\mathsf{RC11}}; \mathsf{eco}^{=}$ *is irreflexive*

4. $\mathsf{rmw} \cap (\mathsf{fr}_e; \mathsf{co}_e) = \emptyset$

5. $[F^{sc}]; (\mathsf{hb}_{\mathsf{RC11}} \cup \mathsf{hb}_{\mathsf{RC11}}; \mathsf{eco}; \mathsf{hb}_{\mathsf{RC11}}); [F^{sc}]$ *is acyclic*

6. *for* $\mathsf{sdep} \in S_E$*,* $\mathsf{sdep} \cup \mathsf{rf}$ *is acyclic*

Taking all relations to be on the execution X*.*

Here the strong RC11 restriction of *acyclic*($\sqsubseteq \cup \mathsf{rf}$) has been updated to use MRD sdep in Condition 6.

6.6 Comparison with other Thin-Air Free models

In this section we present examples that differentiate the Promising Semantics (Kang et al. 2017), WEAKESTMO (Chakraborty and Vafeiadis 2019), and The Event Structures model of Jeffrey and Riely (2016), to our MRD and MRD-C11 models.

6.6.1 A Promising Semantics

First, we show that MRD correctly forbids the out-of-thin-air behaviour, by re-visiting the Coh-CYC litmus test from Chakraborty and Vafeiadis (2019). The test, given below, differentiates Promising and WEAKESTMO: only the latter avoids the outcome $\mathtt{r}_1 = 3$, $\mathtt{r}_2 = 2$ and $\mathtt{r}_3 = 1$.

```
x := 2;                          ||   x := 1;
r1 := x; // 3                    ||   r2 := x; // 2
if(r1 != 2){y := 1}              ||   r3 := y; // 1
                                 ||   if(r3 != 0){x := 3}
```

MRD correctly forbids this outcome: it identifies a dependency on the left-hand thread from the read of 3 from `x` to the write `y := 1`, and on the right-hand thread from the read of 1 from `y` to the write `x := 3`. The desired outcome then has a cycle in dependency and reads-from, and it is forbidden.

Chakraborty and Vafeiadis (2019) ascribe the behaviour to "a violation of coherence or a circular dependency", and include specific machinery to WEAKESTMO that checks for global coherence violations at each step of program execution. These global checks forbid the unwanted outcome.

The Promising Semantics, on the other hand, can make promises that are not sensitive to coherence order, and therefore allows the above outcome erroneously.

6.6.2 WeakestMO

In Coh-CYC, enforcing coherence ordering at each step in WEAKESTMO was enough to forbid the thin-air behaviour, but it is not adequate in all cases. The example below features an outcome that Promising and WEAKESTMO allow, and that MRD-C11 and MRD forbid. It demonstrates that cycles in dependency can arise without violating coherence in WEAKESTMO.

```
                                 if (z != 0) {
z := 1   ||   y := x   ||            r0 := y;
                                     x := r0;
                                     a := r0
                                 }
```

The program is an adaptation[1]of a Java test, where the the unwanted outcome represents a violation of type safety (Lochbihler 2014). Observing the thin-air behaviour where $\mathtt{a} = 1$ in

the adaptation above is the analogue of the unwanted outcome in the original test. If in the end $\mathtt{a} = 1$, then the second branch of the conditional in the rightmost thread must execute. It contains a read of 1 from `y`, and a dependent write of `x := 1`. On the middle thread there is a read of 1 from `x`, and a dependent write of `y := 1`. These dependencies form the archetypal thin-air shape in the execution where $\mathtt{a} = 1$. MRD correctly identifies these dependencies and the outcome is prohibited due to its cycle in reads-from and dependency.

The $\mathtt{a} = 1$ outcome is allowed in the Promising Semantics: a promise can be validated against the write of `x := 1` in the true branch of the right-hand thread, and later switched to a validation with `x :=` $\mathtt{r}_0$ from the false branch, ignoring the dependency on the read of `y`.

In the previous example, Coh-CYC, a stepwise global coherence check caused WEAKESTMO to forbid the unwanted behaviour allowed by Promising, but that machinery does not apply here. WEAKESTMO allows the unwanted outcome, and we conjecture that this deficiency stems from the structure of the model. Dependencies are not represented as a relation at the level of the global axiomatic constraint, so one cannot check that they are consistent with the dynamic execution of memory, as represented by the other relations. Adopting a coherence check in the stepwise generation of the event structure mitigates this concern for Coh-CYC, but not for the test above.

In contrast, MRD does represent dependencies as a relation, allowing us to check consistency with the rf relation here. The axiom that requires acyclicity of $(\mathsf{sdep} \cup \mathsf{rf})$ forbids the unwanted outcome, as desired.

6.6.3 J+R

Like the J+R model, we use event structures to represent program state, and define an algebraic process for finding writes which can always be justified. Unlike the J+R model though, this process happens at each composition in the program with most of the heavy lifting done in the coproduct rule. This modularity makes the semantics easier to evaluate by hand, and by machine, as we shall see later in Chapter 7. Further, MRD explicitly exposes a semantic dependency relation in each execution; this is novel for a thin-air solution.

Another point of comparison is the choice of target for the memory model. MRD and its derivative models are a targetted at modelling C/C++ programs, whereas the authors of J+R chose to focus on a Java model. Java makes different choices about how coherence order is preserved, and is much weaker than C/C++ in this respect. They also do not implement

[1] Riely, Jagadeesan and Jeffrey (2020) provided the precise example presented here. It is based on Fig. 8 of Lochbihler (2014), and its problematic execution under Promising was confirmed with the authors of Promising.

read-modify-write operations, as we have with CAS and FADD for MRD+IMM and MRD+RC11. As we will see in Chapter 7, there are some differences in litmus test out comes as a result of these choices.

6.7 Meta-theory

To analyse the quality of our extensions to IMM and RC11 we show that for each of these models that the MRD-augmented versions are a strict weakening. That is, that any execution allowed under a non-augmented model is also allowed under the MRD-augmented model. Showing this for IMM allows us to show that an MRD-augmented model can be efficient implemented on various hardware targets, as we can re-use the proofs of Podkopaev, Lahav and Vafeiadis (2019). Showing this for RC11 will demonstrate that MRD can be introduced into sensible programming language model for C++, without imposing new restrictions on the compiler or compiler optimisations.

6.7.1 Overview of Proofs

This section offers a high-level explanation of the proofs which will be presented in full in §6.7.3

Theorem (MRD+RC11 is weaker than RC11)**.** *For all executions X in $[\![P]\!]_{RC11}$, X is an execution in $[\![P]\!]_{MRD+RC11}$.*

To show that MRD+RC11 is a usable programming language model, we show that it is at least weaker than RC11 (Lahav et al. 2017), which has several results about for compilation as well as a result that shows common compiler optimisations are valid (Podkopaev, Lahav and Vafeiadis 2019). By showing that any execution X for a program P which is permitted under RC11 is also permitted under MRD+RC11, we can be sure that the same compilation and optimisation results carry forward to MRD+RC11. The full proof of this theorem is presented in §6.7.4.

Theorem (MRD+IMM is weaker than IMM)**.** *For all programs P by the IMM model,*

$$[\![P]\!]_{\mathsf{MRD+IMM}} \supseteq [\![P]\!]_{\mathsf{IMM}}$$

To show that MRD+IMM is implementable over real hardware, we can show that it is a strict weakening of the semantics provided by IMM. So long as for any program P, if there is some execution $X \in [\![P]\!]_{\mathsf{IMM}}$, then there must also be an execution $X' \in [\![P]\!]_{\mathsf{MRD+IMM}}$ such that X and X' are equivalent. With that proof in hand, we can show that MRD+IMM gives the efficient compiler mappings that IMM provides thanks to the proofs of Podkopaev, Lahav and Vafeiadis (2019). The full proof of this theorem is presented in §6.7.5.

6.7.2 DRF-SC

During this work, my collaborators showed that our extension of RC11 preserved the DRF-SC result. This result shows that when the MRD+RC11 model is applied to a C++ program with all SC accesses, the weakening of RC11 into MRD+RC11 does not permit any new data-races. This kind of guarantee for programmers is essential for the model to be usable in practice.

Theorem (MRD+RC11 is DRF-SC)**.** *For a program P, whose atomic accesses are all SC-ordered, if there are no SC-consistent executions with a race over non-atomics, then the outcomes of P under* MRD+RC11 *coincide with those under SC.*

Sketch proof. In the absence of races and relaxed atomics, the no-thin-air guarantee of RC11 is made redundant by the guarantee of happens-before acyclicity shared by RC11 and MRD+RC11. The result follows from this observation, Theorem 6.7.4 and Theorem 4 from Lahav et al. (2017).

6.7.3 Proofs in full

Theorem 6.7.4 (MRD+RC11 is weaker than RC11)**.** *For all executions X in $[\![P]\!]_{RC11}$, X is an execution in $[\![P]\!]_{MRD+RC11}$.*

Proof. Assume an execution X in $[\![P]\!]_{RC11}$. By definition this is a pre-execution in $(E, S, \mathsf{rels}) = \langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$ which is defined as

$$\mathit{freeze}\left(\langle \texttt{Init} \rangle^{\dagger}_{n\ \emptyset\ \lambda\iota.\mathit{freeze}((\langle T_1 \rangle^{\dagger}_{n\ \iota\ \emptyset}) \times \cdots \times (\langle T_m \rangle^{\dagger}_{n\ \iota\ \emptyset}))}\right)$$

for P being a finite set of threads $T_1, \cdots, T_m$. To show that X belongs to $[\![P]\!]_{MRD+RC11}$, it is sufficient to show that X is a pre-execution in $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$ and that it satisfies all the properties in Definition 6.

Both $[\![P]\!]_{RC11}$ and $[\![P]\!]_{MRD+RC11}$ apply the same pre-execution semantics to construct the event structure and the pre-execution set, so X is a pre-execution of $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$.

It remains to show that X satisfies the properties of Definition 9. The only property that differs between the two semantics is the NO-THIN-AIR acyclicity condition. In $[\![P]\!]_{RC11}$, we require $\sqsubseteq \cup \mathsf{rf}$ be acyclic and in $[\![P]\!]_{MRD+RC11}$ we require $\mathsf{sdep} \cup \mathsf{rf}$ to be acyclic, *for some* $\mathsf{sdep} \in S_E$. Thus to show X is in $[\![P]\!]_{\mathsf{MRD+IMM}}$ we just have to show that *there exists* $\mathsf{sdep} \in S_E$ such that $\mathsf{sdep} \subseteq \sqsubseteq$.

We prove this property holds for thread T. The events of X on thread T arise from the interpretation $\langle T \rangle^{\dagger}_{n\ \iota\ \emptyset}$. By the definition of *freeze*, we can find a choice of sdep consistent with $\sqsubseteq$ if for every write, w in T, there exists a $C \vdash w$ such that for all reads $r \in C$, (r, w) in $\sqsubseteq$.

We show by induction that there is always a $\sqsubseteq$-consistent choice of justification for any event of thread T.

- *Case* $\langle - \rangle^{\dagger}_{1\ \iota\ \kappa}$: this evaluates to the empty denotation $\overline{\emptyset}$, which contains no events, so the property trivially holds.
- *Case* $\langle \texttt{skip} \rangle^{\dagger}_{n\ \iota\ \kappa}$: this holds by the induction hypothesis on $\kappa(\iota)$
- *Case* $\langle \texttt{r := x} \rangle^{\dagger}_{n\ \iota\ \kappa}$: interpreting a read results in a read prefix and a coproduct of similar event structures.

  When read prefixing: for any justification edge $C \vdash e$ in the continuation $\kappa(\iota)$, the read rule only removes events from C, preserving the property and adds the read currently being prefixed to the event structure r to the context C. This read r first in program order so $(r, e) \in \sqsubseteq$.

  Then when coproducting justification is strictly extended, preserving the property.
- *Case* $\langle \texttt{x := r} \rangle^{\dagger}_{n\ \iota\ \kappa}$: in the write rule, for any $C \vdash e$ in the continuation $\kappa(\iota)$ the write rule may only remove reads r from C, preserving the property, or through store-forwarding may add the write w which is being prefixed to the event structure to C. As w is being prefixed, it is first in program order – preserving the property.
- *Case* $\langle \texttt{lock} \rangle^{\dagger}_{n\ \iota\ \kappa}$ and *Case* $\langle \texttt{unlock} \rangle^{\dagger}_{n\ \iota\ \kappa}$: the lock and unlock rules does not create any new justification edges, so property holds by the induction hypothesis applied to $\kappa(\iota)$.
- *Case* $\langle P_1 \texttt{;}\ P_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$: similarly to the lock and unlock cases the sequence rule does not directly add any new justification edges, so the property holds by the induction hypothesis on the interpretation of P_1 and P_2.
- *Case* $\langle \textbf{if}\texttt{(}B\texttt{)\{}P_1\texttt{\}\{}P_2\texttt{\}} \rangle^{\dagger}_{n\ \iota\ \kappa}$: as in the sequence rule this holds by the induction hypothesis on the interpretation of P_1 and P_2.
- *Case* $\langle \textbf{while}\texttt{(}B\texttt{)\{}P\texttt{\}} \rangle^{\dagger}_{n\ \iota\ \kappa}$: A while statement does not directly add to justification, so the property holds by the induction hypothesis on the interpretation of P.
- *Case* $\langle P_1 \parallel P_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$: Parallel composition sets the justification to $\emptyset$, which trivially fulfils the property.

□

Theorem 6.7.5 (MRD+IMM is weaker than IMM)**.** *For all programs P by the IMM model,*

$$[\![P]\!]_{\mathsf{MRD+IMM}} \supseteq [\![P]\!]_{\mathsf{IMM}}$$

The executions in $[\![P]\!]_{\mathsf{MRD+IMM}}$ and $[\![P]\!]_{\mathsf{IMM}}$ satisfy the same properties but for the acyclicity check on ar, which differs only in terms of the data dependencies. The only thing to check is that the following property holds:

$$\mathsf{ar}(\mathsf{sdep}) \subseteq \mathsf{ar}(\mathsf{data} \cup \mathsf{ctrl})$$

A first attempt would be show that sdep $\subseteq$ (data $\cup$ ctrl), but this does not hold. Consider the program

$$\mathtt{r}_1 \mathtt{:=} \mathtt{x}\mathtt{;}\ \textbf{fence}\mathtt{;}\ \mathtt{r}_2 \mathtt{:=} \mathtt{y}\mathtt{;}\ \mathtt{y} \mathtt{:=} \mathtt{r}_2$$

Here, the dependency relation sdep would make $\mathtt{y := r}_2$ dependent on $\mathtt{r}_1 \mathtt{:=} \mathtt{x}$, because of Load Forwarding. The data dependency in the IMM model (data) makes $\mathtt{y := r}_2$ dependent on $\mathtt{r}_2 \mathtt{:=} \mathtt{y}$.

We therefore have to prove that sdep $\subseteq$ ar(data $\cup$ ctrl). Since ar contains bob, which orders the reads in program order with the fence, this holds.

We now explain why we use par to calculate dependencies in place of ppo. Consider the program:

$$\mathtt{r}_1 \mathtt{:=} \mathtt{x}\mathtt{;}\ \mathtt{r}_2 \mathtt{:=} \mathtt{x}\mathtt{;}\ \mathtt{y} \mathtt{:=} \mathtt{r}_2$$

If we were to calculate the dependency relation using $\leq$ we would get a dependency from the first read to the write on y due to Load Forwarding, but there is no corresponding ar(data $\cup$ ctrl) edge. By using par to calculate sdep, this dependency is not introduced in the first place – the two reads on x that were in $\leq$ are not in par, and therefore the rule for Load Forwarding does not get triggered.

Definition 10 (Weak event isomorphism)**.** *We define weak event isomorphism ($\approx$) between events with the same memory order and exclusivity, where:*

$$\mathrm{W}\,x\,v \approx \mathrm{W}\,x'\,v' \triangleq x = x'$$
$$\mathrm{R}\,x\,v \approx \mathrm{R}\,x'\,v' \triangleq x = x' \wedge v = v'$$
$$\mathrm{F}(o) \approx \mathrm{F}(o') \triangleq o = o'$$

Otherwise events are not weak event isomorphic.

Definition 11 (Structural isomorphism)**.** *We lift weak isomorphism to configurations. Two*

configurations C and C' are structurally isomorphic, written $C \approx C'$, when there is a function $f : C \to C'$ such that f is a bijection which preserves program order ($\sqsubseteq$) and weak event isomorphism. (i.e. $\forall e.e \approx f(e)$)

Updating preserved program order

We use the IMM notion of preserved ordering, rather than our model's definition. In particular this has an effect on the coproduct, read, and write prefixing rules.

Definition 12 (Pre-ar (par))**.** *In combining MRD with the IMM model, we adjust coproduct lifting so that it respects the thread-local orderings imposed by the IMM model. We collect the edges of* ar *present in the pre-execution, except* data *and* ctrl*, to form a new relation* pre-semantics acyclicity-relation*,* par*.*

$$\mathsf{bob} \cup ([R]; \mathsf{dep}(\emptyset); [W]) \cup ([W_{strong}]; \sqsubseteq; [W]) \qquad \text{(Pre-ar (par))}$$

Read. In the read rule $\leq^X$ is used in the calculation of the "load forwarding" set, LF.

$$\mathrm{LF} = \{(r' : \mathrm{R}\, x\, v) \in C_1 \mid \nexists e.(r, e) \in \mathsf{par} \wedge (e, r') \in \mathsf{par}\}$$

Write. Similarly the "store forwarding" set, SF is updated in the write rule:

$$\mathrm{SF} = \{(r' : \mathrm{R}\, x\, v) \mid \nexists e.(w, e) \in \mathsf{par} \wedge (e, r') \in \mathsf{par}\}$$

Further, the rules for adding a write to the justification context are updated:

1. $\vdash w$
2. $C \setminus \mathrm{SF} \cup \{w\} \vdash e$ if there exists $e' \in C$ s.t. $(w, e') \in \mathsf{par}$
3. $C \setminus \mathrm{SF} \vdash e$ otherwise

Coproduct. In the definition of coproduct we use an upward-closure of justification to enable the lifting of more dependencies. Whenever $C \vdash e$ we define $\uparrow(C)$ as the upward-closed justification set, i.e. $D \vdash e$ if $C \vdash e$, D is a conflict-free lock-free set with $C \subseteq D$, such that for all $e' \in D$ if e'' is an event such that $(e'', e') \in \mathsf{par}$ then $e'' \in D$.

Now we define the coproduct operation. If E_1 is a labelled event structure of the form $(r_1 : \mathrm{R}\, x\, v_1) \bullet E_1'$ and, similarly, E_2 is of the form $(r_2 : \mathrm{R}\, x\, v_2) \bullet E_2'$, the coproduct of event

structures is defined as

$$(E_1, S_1, \vdash_1, \leq_1) + (E_2, S_2, \vdash_2, \leq_2) \triangleq (E_1 + E_2, S_1 \cup S_2, (\vdash_1 \cup \vdash_2 \cup \vdash), \leq)$$

where whenever $\{r_1\} \cup C_1 \vdash_1 (w : \mathrm{W}\, y\, v)$ and $\{r_2\} \cup C_2 \vdash_2 (w' : \mathrm{W}\, y\, v)$ then if the following conditions hold, we have $D' \vdash w_1$ and $D'' \vdash w_2$:

1. there exists a $D' \in \uparrow (C_1)$ that is isomorphic to a $D'' \in \uparrow (C_2)$, that is, there exist $f : D' \to D''$ that is a λ-preserving and par-preserving bijection,
2. there is no event e in D' such that $(r_1, e) \in \mathsf{par}$

Definition 13 (R-downward closure)**.** *Let S^R be the R-downward closure of S.*

$$S^R \triangleq \{e \mid (e, e') \in R \wedge e' \in S\} \cup S$$

Lemma 2 (Structural isomorphism or control dependency)**.** *For all P, C, X, $\mathtt{r}$, v, w, n, ι, and κ, if $w \in C$, $C \subseteq X \wedge (C = C^{\sqsubseteq}) \wedge X \in \langle P \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r} \mapsto v]\ \kappa}$ then:*

$$\exists e \in C.\ (\psi(\mathtt{r}), e) \in \mathsf{ctrl}$$

Or

$$\forall v_i \in V, \exists X', C'.\ \left(C' \subseteq X' \wedge (C' = C'^{\sqsubseteq}) \wedge X' \in \langle P \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r} \mapsto v_i]\ \kappa}\right) \implies C \approx C'$$

when $\kappa(\iota)$ also satisfies this property.

Proof. We prove this by induction over P.

- *Case:* $\langle P \rangle^{\dagger}_{1\ \iota\ \kappa}$ From the rules this evaluates to $\underline{\emptyset}$. $\underline{\emptyset}$ executions, so the property holds trivially.
- *Case:* $\langle \mathtt{skip} \rangle^{\dagger}_{n\ \iota\ \kappa}$ Skip evaluates to its continuation. We then apply the induction hypothesis.
- *Case:* $\langle \mathtt{r}_1 := \mathtt{x} \rangle^{\dagger}_{n\ \iota\ \kappa}$ For execution $X \in \langle \mathtt{r}_1 := \mathtt{x} \rangle^{\dagger}_{n\ \iota\ \kappa}$, the set of $\sqsubseteq$-downclosed configurations are of the form $\{r : \mathrm{R}\, x\, v_r\} \cup C$ where C is a $\leq$-downclosed configuration of $\langle P \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r}_1 \mapsto v_r]\ \kappa}$. If there is an event $e \in C$ with $(\psi(\mathtt{r}), e) \in \mathsf{ctrl}$, then Proposition 3 implies $e \in r \cup C$, as required.

  Otherwise, case split on $\mathtt{r}_1 = \mathtt{r}$. If true, then for C, a $\leq$-downclosed configuration of $\kappa\iota.\rho[\mathtt{r}_1 \mapsto v_r]$, we have $\{r : \mathrm{R}\, x\, v_r\} \cup C$ is a $\leq$-downclosed configuration of $\langle \mathtt{r}_1 := \mathtt{x} \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r}_1 \mapsto v]\ \kappa}$

and $\langle \mathtt{r}_1 \texttt{ := } \mathtt{x} \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r}_1 \mapsto v']\ \kappa}$. This case is satisfied by observing that structural isomorphism is reflexive.

If $\mathtt{r}_1 \neq \mathtt{r}$, then for C, a $\leq$-downclosed configuration of $\kappa\iota.\rho[\mathtt{r}_1 \mapsto v]$, we have $\{r : \mathrm{R}\, x\, v_r\} \cup C$ is a $\leq$-downclosed configuration of $\langle \mathtt{r}_1 \texttt{ := } \mathtt{x} \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r}_1 \mapsto v]\ \kappa}$. We invoke the inductive hypothesis to find structurally isomorphic configuration C' of $\kappa\iota.\rho[\mathtt{r}_1 \mapsto v']$, and this gives us configuration $r \cup C'$ of $\langle \mathtt{r}_1 \texttt{ := } \mathtt{x} \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r}_1 \mapsto v']\ \kappa}$. Finally, the definition of $C \approx C'$ implies the existence of a weak-event-isomorphic function f relating the events of C and C'. We extend this, mapping r in $r \cup C$ to r in $r \cup C'$ to establish structural isomorphism between the two.

- *Case:* $\langle \mathtt{x} \texttt{ := } M \rangle^{\dagger}_{n\ \iota\ \kappa}$ The write case is similar to the read case, but the value of the appended write event may depend on r. Note that two writes of the same location are structurally isomorphic if their location matches, but their written value does not have to match.

- *Case:* $\langle P_1 \texttt{;}\ P_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$ This case is satisfied by the application of the inductive hypothesis on P_1 and P_2.

- *Case:* $\langle \textbf{if}(B)\{P_1\}\{P_2\} \rangle^{\dagger}_{n\ \iota\ \kappa}$ We case split on $\mathtt{r} \in B$. If it is not, then the interpretation of B under $\rho[\mathtt{r}_1 \mapsto v]$ and $\rho[\mathtt{r}_1 \mapsto v']$, and the property is satisfied by the inductive hypothesis.

  If $\mathtt{r} \in B$, then we add $\psi(\mathtt{r})$ to γ. We have $(\psi(\mathtt{r}), w) \in \mathsf{ctrl}$ by the write rule applied over $w \in C$ as required.

- *Case:* $\langle \textbf{\textit{fence}}^{o_F} \rangle^{\dagger}_{n\ \iota\ \kappa}$ We follow the argument from the read case.

- *Case:* $\langle \textbf{while}(B)\{P\} \rangle^{\dagger}_{n\ \iota\ \kappa}$ We apply the same argument as the if case.

- *Case:* $\langle \mathtt{r}_1 \texttt{ := } \textbf{\textit{CAS}}^{o_R, o_W}_{o_{RMW}}(\mathtt{x}, M_1, M_2) \rangle^{\dagger}_{n\ \iota\ \kappa}$ This case follows similar arguments to the read and write cases.

- *Case:* $\langle \mathtt{r}_1 \texttt{ := } \textbf{\textit{FADD}}^{o_R, o_W}_{o_{RMW}}(\mathtt{x}, M) \rangle^{\dagger}_{n\ \iota\ \kappa}$ This case follows similar arguments to the read and write cases.

□

Lemma 3 (configurations have equal write values or data dependency)**.** *For a down-closed configurations* $C \in \langle P \rangle^{\dagger}_{n\ \iota.\rho\ \emptyset}$*, and* $C' \in \langle P \rangle^{\dagger}_{n\ \iota.\rho[\mathtt{r} \mapsto v]\ \emptyset}$ *Where* $C \approx C'$ *through bijection* f*,*

$$\forall (w : \mathrm{W}\, x\, m) \in C.\, (\mathit{value}(f(w)) = m) \vee ((\psi(\mathtt{r}), w) \in \mathsf{data} \cup \mathsf{ctrl}) \tag{24}$$

Proof. We prove this by induction over P.

- *Case:* $\langle P \rangle^{\dagger}_{1\ \iota\ \kappa}$ From the rules this evaluates to $\underline{\emptyset}$. $\underline{\emptyset}$ contains no writes, so the property holds trivially.
- *Case:* $\langle \texttt{skip} \rangle^{\dagger}_{n\ \iota\ \kappa}$ Skip evaluates to its continuation. We then apply the induction hypothesis.
- *Case:* $\langle \texttt{r}_1 \texttt{ := x} \rangle^{\dagger}_{n\ \iota\ \kappa}$Interpretation of a read does not add any write events to the configurations of $\kappa(\iota)$, so the property holds by the induction hypothesis.
- *Case:* $\langle \texttt{x := } M \rangle^{\dagger}_{n\ \iota\ \kappa}$ The appended write event must be part of any downclosed configuration, in particular $C \in \langle \texttt{x := } M \rangle^{\dagger}_{n\ \iota.\rho\ \kappa}$, and $C' \in \langle \texttt{x := } M \rangle^{\dagger}_{n\ \iota.\rho[\texttt{r} \mapsto v]\ \kappa}$ where $C \approx C'$ through bijection f.

  There are three cases for M:

  - where M is a value n and there is a read in γ from $\psi(\texttt{r}_1)$. In this case we create a ctrl-dependency, as required.
  - If M is a value and there is no read in γ from $\psi(\texttt{r}_1)$, then we are done: $f(w) = w$.
  - Otherwise, suppose that $\texttt{r}_1 = \texttt{r}$, then the write rule implies $(\psi(\texttt{r}), w) \in$ data, as required. If $\texttt{r}_1 \neq \texttt{r}$, then $\rho(\texttt{r}_1) = \rho[\texttt{r} \mapsto v](\texttt{r}_1)$, as required.
- *Case:* $\langle P_1 \texttt{ ; } P_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$ This case is satisfied by the application of the inductive hypothesis on P_1 and P_2.
- *Case:* $\langle \textbf{if}(B)\{P_1\}\{P_2\} \rangle^{\dagger}_{n\ \iota\ \kappa}$ We case split on $\texttt{r}_1 \in B$. If it is not in B then we're done, because we do not add $\psi(\texttt{r}_1)$ to γ. If it is, then we add $\psi(\texttt{r}_1)$ to γ which is the beginning of a new ctrl-dependency, as in the write rule.
- *Case:* $\langle \boldsymbol{fence}^{o_F} \rangle^{\dagger}_{n\ \iota\ \kappa}$ We follow the argument from the read case.
- *Case:* $\langle \textbf{while}(B)\{P\} \rangle^{\dagger}_{n\ \iota\ \kappa}$ Again, we apply the induction hypothesis.
- *Case:* $\langle \texttt{r}_1 \texttt{ := } \boldsymbol{CAS}^{o_R, o_W}_{o_{RMW}}(\texttt{x}, M_1, M_2) \rangle^{\dagger}_{n\ \iota\ \kappa}$ This case follows similar arguments to the read and write cases.
- *Case:* $\langle \texttt{r}_1 \texttt{ := } \boldsymbol{FADD}^{o_R, o_W}_{o_{RMW}}(\texttt{x}, M) \rangle^{\dagger}_{n\ \iota\ \kappa}$ This case follows similar arguments to the read and write cases.

□

Proposition 1 (append preserves excution suffixes.)**.** *For all executions* $X \in \langle T_1; T_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$*, where* $T_1; T_2$ *is a single-threaded program, there exists* $X_2 \in \langle T_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$*, such that* $X_2 \subseteq X$*.*

Proof. This follows by induction on T_1. Event prefixing extends the executions of the continuation. All other rules leave the executions in place. □

Proposition 2 (intervening ctrl implies ctrl)**.** *For* $X \in \langle P \rangle^{\dagger}_{n\ \iota\ \kappa}$*, read event* $r \in X$*, write event* $w \in X$*, and event* $e \in X$*. If* $(r, e) \in \mathsf{ctrl}$ *and* $(e, w) \in \sqsubseteq$ *then* $(r, w) \in \mathsf{ctrl}$

Proof. In the interpretation of a program containing an **if**-statement, γ is updated to contain the relevant reads. These reads are always passed through to the continuation in the rest of the rules. In the interpretation of a write w, the reads of γ are used to construct a ctrl dependency. □

Proposition 3 (ctrl persists on apend)**.** *For* T_1*,* T_2*, read* r*, and write* w *with* $\mathsf{rels}_2 \in \langle T_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$*, and* $\mathsf{rels} \in \langle T_1; T_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$*, we have* $(r, w) \in \mathsf{rels}_2.\mathsf{ctrl}$ *implies* $(r, w) \in \mathsf{rels}.\mathsf{ctrl}$

Proof. Induction over T_1, observing no rule removes edges from ctrl. □

Lemma 4 (Justifying writes imply ar-successor justifying read)**.** *For all* P *where* $C \vdash e \in \langle P \rangle^{\dagger}_{n\ \iota\ \kappa}$*, with a write* $w \in C$ *there exists* r *such that* $(w, r) \in \mathsf{par}$ *and there exist* C_2*,* P_1 *and* P_2 *such that* $P = P_1; P_2$*,* $C_2 \vdash e \in \langle P_2 \rangle^{\dagger}_{n\ \iota\ \kappa}$*, and* $r \in C_2$*.*

Proof. We prove by induction over P that events added to the justification context are reads, or there is an ar edge to an event previously within the justification context.

- $\langle - \rangle^{\dagger}_{1\ \iota\ \kappa}$ this rule evaluates to the empty denotation, where there are no justifications, so the property trivially holds.
- $\langle \texttt{skip} \rangle^{\dagger}_{n\ \iota\ \kappa}$ this rule evaluates to the continuation $\kappa(\iota)$, property holds by the induction hypothesis.
- $\langle \texttt{r := x} \rangle^{\dagger}_{n\ \iota\ \kappa}$ this rule adds only reads to the justification contexts, so the property holds.
- $\langle \texttt{x := r} \rangle^{\dagger}_{n\ \iota\ \kappa}$ this rule adds writes the the justification context only if there is an event in in that context following in ar.
- The FADD and CAS rules are similar.
- The sequence, while, fence and if rules adopt the justification of the continuation without change.

Now, any justifying write event corresponds to a sequence of ar-related events terminating in a justifying read, as required. □

Proof of Theorem 6.7.5. Assume an execution X in $[\![P]\!]_{\mathsf{IMM}}$. By definition this is a pre-execution in $(E, S, \mathsf{rels}) = \langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$ which is defined as

$$\mathit{freeze}(\langle \texttt{Init} \rangle^{\dagger}_{n\ \emptyset\ \lambda\iota.\mathit{freeze}((\langle T_1 \rangle^{\dagger}_{n\ \iota\ \emptyset}) \times \cdots \times (\langle T_m \rangle^{\dagger}_{n\ \iota\ \emptyset}))})$$

for P being a finite set of threads $T_1, \cdots, T_m$. To show that X belongs to $[\![P]\!]_{\mathsf{MRD+IMM}}$, it is sufficient to show that X is a pre-execution in $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$ and that it satisfies all the properties in Definition 6.

Both $[\![P]\!]_{\mathsf{IMM}}$ and $[\![P]\!]_{\mathsf{MRD+IMM}}$ apply the same pre-execution semantics to construct the event structure and the pre-execution set, so X is a pre-execution of $\langle P \rangle^{\dagger}_{n\ \emptyset\ \emptyset}$.

It remains to show that X satisfies the properties of Definition 6. The only property that differs between the two semantics is the acyclicity condition on ar. In $[\![P]\!]_{\mathsf{IMM}}$, we require $\mathsf{ar}_X(\mathsf{data} \cup \mathsf{ctrl})$ to be acyclic and in $[\![P]\!]_{\mathsf{MRD+IMM}}$ we require $\mathsf{ar}_X(\mathsf{sdep})$ to be acyclic, *for some* $\mathsf{sdep} \in S_E$. Thus to show X is in $[\![P]\!]_{\mathsf{MRD+IMM}}$ we just have to show that *there exists* $\mathsf{sdep} \in S_E$ such that $\mathsf{sdep} \subseteq \mathsf{ar}_X(\mathsf{data} \cup \mathsf{ctrl})$.

We prove this property holds for thread T. The events of X on thread T arise from the interpretation $\langle T \rangle^{\dagger}_{n\ \iota\ \emptyset}$. By the definition of *freeze*, we can find a choice of sdep consistent with ar_X if for every write, w in T, there exists a $C \vdash w$ such that for all reads $r \in C$, (r, w) in ar_X.

We try to construct a justification edge where there is no ar_X: suppose there is a write $w \in X$ from thread T where all configurations that justify w, and that are made only of $\sqsubseteq$ predecessors of w, contain a read event that creates a dependency that is not in ar_X. The set of justifications of w is not empty: the write rule adds a justification for w, and the rest of the rules leave at least one $\sqsubseteq$ consistent justification in place.

We choose one $\sqsubseteq$-consistent justification $C \vdash w$, with read $r \in C$ such that $(r, w) \in \sqsubseteq$ and $(r, w) \notin \mathsf{ar}_X$.

We define I to be a set of pairs, containing r and all $\sqsubseteq$-following read events which are $\sqsubseteq$-before w paired with the justification present as a result of their construction. This set is finite and totally ordered by $\sqsubseteq$, so we write it as a sequence of $k+1$ elements: $(r, \vdash), (r_1, \vdash_1), \ldots, (r_k, \vdash_k)$.

Now we show by induction that for every element $r_i, \vdash_i$ of I, $r_i \vdash_i w \implies (r_i, w) \in \mathsf{ar}_X(\mathsf{data} \cup \mathsf{ctrl})$. We take r_k as the base case, and in the inductive step we assume the property holds for all successors in the sequence.

Inductive step: r_i. Read event r_i is constructed by the rules for load, FADD or CAS in T. Take $T_1; \texttt{r := } T_{r_i}; T_2 = T$, such that T_{r_i} is this load, FADD or CAS. Apply Proposition 1 to get $X_2 \in \langle T_2 \rangle^{\dagger}_{n\ \iota\ \emptyset}$ containing event w. Let A_{r_i} be the set containing w and its $\sqsubseteq$-predecessors in X_2

and apply Lemma 2 to T_2, X_2 and A_{r_i}. Now we case split on whether there is an event $e \in A_{r_i}$ such that $(\psi(\mathtt{r}), e) \in \mathsf{ctrl}$.

In the first case, we have $(r_i, w) \in \mathsf{ctrl}$ by Proposition 2. Further, by Proposition 3, we have $(r, w) \in \mathsf{rels}.\mathsf{ctrl}$ for $\mathsf{rels} \in \langle T \rangle^{\dagger}_{n\ \iota\ \emptyset}$ and $(r, w) \in \mathsf{ar}_X$, as required.

In the second case, Lemma 2 provides structural isomorphism between A_{r_i} and $A_{r_i^v}$, the corresponding set of events following conflicting read r_i^v for value $v \in V$. We have an event $w_{r_i^v} \in A_{r_i^v}$ that is structurally isomorphic to w. Proposition 3 implies either a data or ctrl dependency from r_i to w, as required, or $w \simeq w_{r_i^v}$.

We have a justification edge from r_i to w, so coproduct did not lift the justification of w above the read event r_i. We have isomorphism of w and $w_{r_i^v}$ for all $v \in V$. There must exist C_{r_i} and $C_{r_i^v}$ such that $r_i \in C_{r_i}$, $r_i^v \in C_{r_i^v}$, $C_{r_i} \vdash_i w$ and $C_{r_i^v} \vdash_i w_{r_i^v}$. Every element of $\uparrow(C_{r_i})$ is a subset of A_{r_i} and similarly each element of $\uparrow(C_{r_i^v})$ is a subset of $A_{r_i^v}$.

Appealing to the definition of coproduct, one of the following must hold:

- There is an event $e \in \uparrow(A_{r_i})$ such that $(r_i, e) \in \mathsf{par}$.
- There do not exist upward closures of C_{r_i}, $C_{r_i^v}$ that are label isomorphic.
- There do not exist upward closures of C_{r_i}, $C_{r_i^v}$ that are isomorphic in the par relation.

Case 1. We case split on whether e is a read or a write. For read e, we apply the inductive hypothesis and we have that $(r_i, w) \in \mathsf{par}^+$ as required. For write e we apply Lemma 4 to find par-later read e' with $e' \vdash_j w$. Again we apply the inductive hypothesis to get $(r_i, w) \in \mathsf{par}^+$ as required.

Case 2. In this case, all upward closures include an event that is not label isomorphic with its mapped event. We take upward closures $D_{r_i} \in \uparrow(C_{r_i})$ and $D_{r_i^v} \uparrow (C_{r_i^v})$, and we have event $e \in D_{r_i}$ and isomorphism f such that $\lambda e \neq \lambda(f(e))$. By lemma 2 and lemma 3 we have that e and $f(e)$ must be writes and $(r_i, e) \in \mathsf{ctrl} \cup \mathsf{data}$. We apply Lemma 4 to write e to find a par-later read r' with $r' \vdash_j w$. We apply the inductive hypothesis to get $(r_i, w) \in \mathsf{par}^+$ as required.

Case 3. In this case, all upward closures include an event where the isomorphism does not preserve par. We take upward closures $D_{r_i} \in \uparrow(C_{r_i})$ and $D_{r_i^v} \uparrow (C_{r_i^v})$ and we have event $e \in D_{r_i}$ and isomorphism f such that

$$\exists e' \in D_{r_i}.(e, e') \in \mathsf{par} \not\Leftrightarrow (f(e), f(e')) \in \mathsf{par} \tag{25}$$

By Case 2 above, we have that pairs $(e, f(e))$ and $(e', f(e'))$ are isomorphic. Moreover, by lemma 2 we have that all reads are isomorphic, so read-to-read relation casdep is preserved by the isomorphism, as are all other components of par. This contradicts 25, completing this case.

Base case. We follow the same argument as in the inductive case, but by Lemma 4, there can be no writes in Cases 1, 2 and 3 above. □

6.8 Conclusion

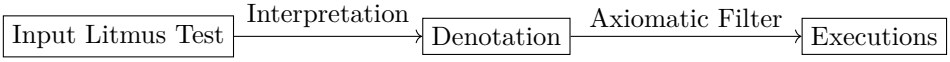
We have extended the MRD semantics to support hardware and software features required by the existing C++ memory model, and by IMM. The former shows that MRD can be used to create an industrial-strength thin-air free memory model which can describe C++ program execution without admitting problematic OOTA behaviour. The latter shows us that MRD is weak enough that it can be implemented on hardware efficiently without necessitating additional fences to be emitted by a compiler. We also see how the approach of MRD is flexible to interact with any axiomatic system: by defining sdep as a relation and exposing executions which are compatible with standard axiomatic memory models, MRD provides a framework for extending any axiomatic model to be thin-air free without requiring restrictive axioms like $acyclic(\mathsf{rf} \cup \mathsf{po})$. In the next chapter, we will evaluate MRD empirically by evaluating a large corpus of litmus tests through MRD, MRD+IMM, and MRD+RC11.

Chapter 7

Mechanical Analysis of Modular Relaxed Dependencies

MRD is the first weak memory model to solve the thin-air problem for C++ atomics that has a tool for automatically evaluating litmus tests. Our tool, MRDer, evaluates litmus tests under the base model, RC11 augmented with MRD, and IMM augmented with MRD. It has been used to check the result of every litmus test in this paper, together with many tests from the literature, including the Java Causality Test cases (Pugh 2004), and all tests from Batty et al. (2015); Chakraborty and Vafeiadis (2019); Jeffrey and Riely (2016); Kang et al. (2017); Lahav et al. (2017); Pichon-Pharabod and Sewell (2016); Podkopaev, Lahav and Vafeiadis (2019).

MRDer is built in OCaml, tightly following the formal definitions presented in previous sections. Evaluation takes place in two steps following the structure of the global semantics: first a program is interpreted into an OCaml data type representing the event structure, then candidate executions are extracted and filtered according to the axioms of the particular model.



The denotation stage builds a structure which contains semantic dependencies along with a set of candidate executions. These candidate executions are then filtered by an axiomatic memory model. We have a simplistic model which only constrains coherence (MRD); an augmented hardware model (MRD+IMM); and a full-fat C/C++ model (MRD+RC11). The coherence-only model is defined in §5, and the augmented models are both defined in §6. MRDer evaluates litmus tests under the base MRD model, the augmented MRD+IMM and MRD+RC11 models, as well as the un-augmented IMM and RC11 models for comparison. We have collected a large

suite of litmus tests from across the academic literature to evaluate MRD in the context of its competitors and previous work on weak memory consistency. This mechanical validation, along with the metatheory proved in §6, gives two substantial arguments in favour of MRD as a good definition of sdep; and MRD+RC11 as a strong candidate for the future definition of C/C++.

7.1 MRDer at scale

In MRD, it is the calculation of justification that draws in information from other executions. This mechanism is localised, it avoids making choices about the execution that prune behaviours, and it does not require backtracking. MRD acts in a "bottom-up" fashion, and modularity ensures that justifications drawn from the continuation need not be recalculated. These properties have supported the development of MRDer: automation of the model requires only a single pass across the program text to construct the denotation. While we do not have any formal result that compares the complexity of MRD with other models, our experience from developing MRDer indicates that it is easier to implement.

Consider, for example, the mechanism of promises in A Promising Semantics, Kang et al. (2017). At every machine step in Promising, one can make any number of τ transitions after which the program must be executed forwards to search for an execution which validates the current set of promises. Each of the decisions made while executing promising creates combinatorially more possible traces to a given execution, and each of those choices should be fully explored by an automated tool. Sadly, the authors of Promising have not provided a tool, and we would hazard that building a tool to automatically evaluate Promising over a set of litmus tests would be non-trivial to say the least. Preliminary research on this suggests that the Promise mechanism is un-decidable without substantial constraints, Abdulla et al. (2020).

7.1.1 Related tools

While MRDer is the first tool for a thin-air free C++ memory model, it is not the first tool for thin-air free models, or for memory models in general. The first tool for a thin-air free model is PrideMM, which was presented in §4, and many tools exist for evaluating non thin-air free memory models – though the leading tool is Herd (Alglave, Maranget and Tautschnig 2014). Each of these tools achieve different things in different ways. PrideMM provides an API for building higher-order equations in Second Order logic, which is then applied to express the J+R event structures model (Jeffrey and Riely 2016) in §4.4.4. Models written for PrideMM must fit into a single equation which can be calculated over an input program. The architecture of

PrideMM is a good fit for expressing J+R: where the interpretation over program syntax is simple and a complex equation explores the structure generated by the interpretation to produce a set of executions. Conversely, this approach would be a poor fit for MRD whose calculation is a complex interpretation over program syntax to generate an artefact which can be trivially unwrapped to find the valid executions. Herd provides a language for building axiomatic memory models and evaluating all executions permitted on a given litmus test, but is tied to a very specific set of assumptions about the structure of an axiomatic memory model, and extension to incorporate a calculation along the lines of MRD would be no better than building MRDer from scratch.

7.2 Tests from the literature

In this section we enumerate many tests from various pieces of literature and present tables of results for our semantics, as well as base and MRD-augmented RC11 and IMM semantics. Each table has a column for these, $[\![P]\!]$, $[\![P]\!]_{\mathsf{RC11}}$, $[\![P]\!]_{\mathsf{MRD+RC11}}$, $[\![P]\!]_{\mathsf{IMM}}$, and $[\![P]\!]_{\mathsf{MRD+IMM}}$ – which respectively represent our base MRD semantics (§5), the base RC11 semantics (Lahav et al. 2017), RC11 augmented with MRD's sdep (Definition 9), the base IMM semantics (Podkopaev, Lahav and Vafeiadis 2019), and finally IMM augmented with MRD's sdep (Definition 6). Each test specifies a program, and an assertion: ✔ indicates that a given model agrees with the assertion, and ✘ indicates a disagreement. Each row of tests is evaluated in a single pass of MRDer, and the execution time is given in the final column of the results tables. These tests were performed on an i5-5250U processor with 16 GB of memory.

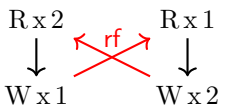
7.2.1 Java Causality Test Cases

The Java Causality Test Cases (Pugh 2004) (JCTCs) are a set of litmus tests which demonstrate how compiler optimisations in Java can cause weak memory behaviours. A good Java memory model would agree with the outcome specified in each test, allowing some weak executions permitted by compiler optimisations and forbidding the executions that are nonsensical in Java. Many of the optimisations mentioned in the JCTCs can be found in C/C++ compilers as well, and as such these results are great validation for this as a C/C++ semantics, as well as a potential starting-point for a thin-air free Java semantics.

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
JCTC 1	✔	✘	✔	✔	✔	0.16
JCTC 2	✔	✘	✔	✘	✔	0.23
JCTC 3	✔	✘	✔	✘	✔	0.88
JCTC 4	✔	✔	✔	✔	✔	2.12
JCTC 5	✔	✔	✔	✔	✔	0.14
JCTC 6	✔	✘	✔	✘	✔	3.40
JCTC 7	✔	✘	✔	✔	✔	0.16
JCTC 8	✔	✘	✔	✘	✔	0.23
JCTC 9	✔	✘	✔	✘	✔	0.58
JCTC 10	✔	✔	✔	✔	✔	0.94
JCTC 11	✔	✘	✔	✔	✔	427.06
JCTC 13	✔	✔	✔	✔	✔	0.05
JCTC 14	✔	✔	✔	✔	✔	12.65
JCTC 15	✔	✔	✔	✔	✔	24.27
JCTC 16	✔	✘	✘	✘	✘	0.55
JCTC 17	✘	✘	✘	✘	✔	3.85
JCTC 18	✘	✘	✘	✘	✔	3.95
JCTC 19	✘	✘	✘	✘	✘	0.13
JCTC 20	✘	✘	✘	✘	✘	0.11

In the table above we enumerate each of the JCTCs with the exception of JCTC 12 which relies on address dependencies which are not modelled in this work.

One of the more interesting rows is JCTC 16: it is allowed by our base model, but not by any of the C/C++ axiomatic models, including the MRD augmented versions. JCTC 16, printed in Figure 48, presents one of the differences between C/C++ and Java: unlike C++, Java does not have a global coherence ordering. This constraint lies entirely within the axiomatic C++ portion of the memory model, though – it is not a limitation of MRD. A forbidden C++ execution is drawn below.



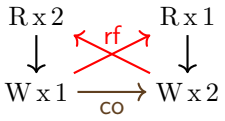
There is a cycle in $\mathsf{po} \cup \mathsf{rf}$, which means that this execution is not observable in RC11, but this does not explain why it would be disallowed in the augmented MRD +RC11 model. Instead this is a demonstration of how our changes are orthogonal to the rest of the memory model. Despite

Initially: $\mathtt{x} = 0$

Thread 1	**Thread 2**
$\mathtt{r}_1$:= x;	$\mathtt{r}_2$:= x;
x := 1	x := 2

Figure 48: Java Causality Test Case 16. $\mathtt{r}_1 = 2 \wedge \mathtt{r}_2 = 1$ must be allowed by a Java memory model.

the $\mathsf{po} \cup \mathsf{rf}$ cycle this execution would be forbidden by another axiom in RC11. RC11 requires all writes to the same location to be totally ordered by co, so in this case there must be one of $\mathrm{W}\,x\,1 \xrightarrow{\mathsf{co}} \mathrm{W}\,x\,2$ or $\mathrm{W}\,x\,2 \xrightarrow{\mathsf{co}} \mathrm{W}\,x\,1$ present. Take the following example with $\mathrm{W}\,x\,1 \xrightarrow{\mathsf{co}} \mathrm{W}\,x\,2$.



Here there is a cycle in $\mathsf{rf} \cup \mathsf{po} \cup \mathsf{co}$, which is forbidden by the Coherence axiom of RC11. A similar cycle would be present for the alternative choice of $\mathrm{W}\,x\,2 \xrightarrow{\mathsf{co}} \mathrm{W}\,x\,1$.

With minor modification to include coherence and allow thread-inlining optimisations, MRD could be extended to pass all of the Java Causality Test Cases. Currently, the closest semantics for the JCTCs is Jeffrey and Riely (2016), which disagrees with test cases 3, 7 and 11. We note that it is not totally clear in their presentation how they interpret programs with join operations, as there is no constructor for join in their input language. That being said, it would not be a perfect fit for the Java programming specification to adopt a MRD-based memory model, as the current specification is operational (Manson, Pugh and Adve 2005); this is unlike the axiomatic model of C and C++ where our dependency relation can be integrated seamlessly.

Another interesting observation from this test suite is the execution time of JCTC 11 which is an outlier compared to the other tests, at 427.06s. This is down to a set of sequential reads, who grow the event structure combinatorially. The event structure for JCTC 11 has over 50 events, which results in large complex coproduct operations. The performance is an artefact of MRDer's development, it was not written with performance in mind – instead the code closely reflects the mathematical definitions of the model. For most litmus tests this poor scaling is acceptable, as litmus tests are typically very small programs.

7.2.2 Common Hardware Litmus Tests

Next we shall look at a set of tests which exemplify how solving the thin-air problem can be isolated from the existing solutions for memory model research. This section presents a set of common litmus tests which reveal hardware behaviours present on weakly ordered architectures, in this case they're focused on POWER – but most of these tests have analogues for ARM too. The formalisation of the POWER memory model and these litmus tests were originally presented by Sarkar et al. (2011). The tests describe how synchronisation on POWER works, what weak patterns are allowed and which are forbidden.

To compile down to to efficient hardware instructions for loads and stores, a C++ model must preserve the weak semantics of the hardware otherwise it would need to include fences to preserve some additional ordering semantics implied by the language. It is therefore critical for a C/C++ memory model to be weaker than the hardware model for POWER and ARM, and as these POWER tests do not interact with dependencies our implementation of the C++ memory model must preserve the weak executions too. As such, our model and the other axiomatic C/C++ models should all agree with the POWER hardware litmus tests.

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
2+2W	✔	✔	✔	✔	✔	53.03
3.LB	✔	✘	✔	✔	✔	6.69
CoRR	✔	✔	✔	✔	✔	0.06
CoRW	✘	✔	✔	✔	✔	2.42
CoWR	✘	✔	✔	✔	✔	1.80
CoWW	✘	✔	✔	✔	✔	0.29
IRIW	✔	✔	✔	✔	✔	1.67
IRRWIW	✔	✔	✔	✔	✔	3.85
IRWIW	✔	✔	✔	✔	✔	8.27
ISA2	✔	✔	✔	✔	✔	0.82
LB	✔	✘	✔	✔	✔	0.47
MP	✔	✔	✔	✔	✔	0.12
PPO000-019	✔	✔	✔	✔	✔	0.53
R	✔	✔	✔	✔	✔	0.11
RWC	✔	✔	✔	✔	✔	0.38
S	✔	✔	✔	✔	✔	5.54
SB	✔	✔	✔	✔	✔	0.11
WRC	✔	✔	✔	✔	✔	0.80
WRR+2R	✔	✔	✔	✔	✔	0.56
WRW+2W	✔	✔	✔	✔	✔	1.07
WRW+WR	✔	✔	✔	✔	✔	0.66
WWC	✔	✔	✔	✔	✔	2.05
W+RWC	✔	✔	✔	✔	✔	0.81
Z6.0	✔	✔	✔	✔	✔	1.69
Z6.1	✔	✔	✔	✔	✔	2.17
Z6.2	✔	✔	✔	✔	✔	3.54
Z6.3	✔	✔	✔	✔	✔	0.96
Z6.4	✔	✔	✔	✔	✔	1.05
Z6.5	✔	✔	✔	✔	✔	1.06

One interesting exception are the CoRW, CoWR, and CoWW tests for the base semantics. These are coherence tests and they expose the simplistic overly strong coherence axiom present in the base semantics, but replaced in the augmented semantics. When we use MRD to generate a sdep relation, and incorporate that into RC11 or IMM we find that we agree on all tests here.

This is also further validation that we do not unduly forbid program executions which must be permitted in a C/C++ implementation, which we have proven in §6 with Theorems 6.7.5 and 6.7.4.

Bridging the Gap between Programming Languages

The IMM model, presented by Podkopaev, Lahav and Vafeiadis (2019), is designed to be an "Intermediate Memory Model", between programming languages and hardware. It permits all the behaviours required for efficient implementation to hardware for RISC-V, ARMv7, ARMv8, x86-TSO and POWER. As with the POWER tests these tests provide validation that our model can be mapped to hardware behaviours.

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
MP+rel+acq	-	✔	✔	✔	✔	0.12
FADD	-	✔	✔	✔	✔	27.56
RMW-atomicity	-	✔	✔	✔	✔	0.24
LB+rel+rel	-	✔	✔	✔	✔	0.12
LB+acq+acq	-	✔	✔	✔	✔	0.12
LB+addr+rel	-	-	-	-	-	-
rfi-preservation	-	✘	✘	✔	✔	8.65
Enforcing detour	✔	✔	✔	✔	✔	6.97
IRIW	-	✔	✔	✔	✔	2.82
PSC	-	✔	✔	✔	✔	0.78
ARM-FADD	-	✔	✔	✔	✔	22.91
LB	✔	✘	✔	✔	✔	0.23

The toy language used in MRD does not support address dependencies, and operations like fences and read-modify-writes are only considered in the augmented models – as extra axioms are required to give these operations any semantic force.

7.2.3 Tests from other Thin-Air Free Models

We use these tests to validate our model in the context of other thin-air free memory models, like Promising and WeakestMO. These comparisons provide an interesting point to compare MRD to its competitors.

Repairing Sequential Consistency in C/C++11

RC11 provided the simplest mechanism for forbidding OOTA behaviour, but at the cost of forbidding important compiler optimisations.

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD\text{-}C11}}$	$[\![P]\!]_{\mathrm{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
RWC+syncs	-	✔	✔	✔	✔	0.61
W+WRC	-	✔	✔	✔	✔	1.27
LB+deps	✔	✔	✔	✔	✔	0.46
LB	✘	✔	✘	✘	✘	0.48

The No-Thin-Air axiom of RC11 ($\mathit{acyclic}(\mathsf{po} \cup \mathsf{rf})$) is deliberately not included in our augmented models, as we consider the restriction imposed by RC11 to be overly strong. The $\mathit{acyclic}(\mathsf{po} \cup \mathsf{rf})$ restriction is exactly the axiom that LB probes, where a weak execution has exactly that cycle. It can be exhibited on hardware which permits Load/Store re-ordering, such as NVIDIA GPUs. The No-Thin-Air axiom would forbid the behaviours visible in NVIDIA hardware and efficient compilation would require a fence between loads and stores to prevent re-ordering. Thus the No-Thin-Air axiom is undesirable in a C++ model – indeed MRD omits it hence our disagreement with LB. Further note that many of the tests in this paper refer to SC reads and writes, the IMM implementation of this model (that we use) does not cover SC loads and stores. Recent work (Moiseenko et al. 2019) has extended IMM to include SC loads and stores, so as future work we'll incorporate the definition changes.

A Promising Semantics for Relaxed-Memory Concurrency

These tests come from Kang et al. (2017) who presented Promising. Promising seeks to be an operational model which forbids out-of-thin-air executions by allowing exploration through the possible future executions of the programs to find invariants which can be used to explain the behaviour of a particular thread.

Initially: $\mathtt{x} = \mathtt{y} = 0$

Thread 1		**Thread 2**
$\mathtt{r}_1 := \mathtt{x};$		
$\mathtt{y} := 1;$	$\Vert$	$\mathtt{r}_3 := \mathtt{y};$
$\mathtt{r}_2 :=^{acq} \mathtt{y};$	$\Vert$	$\mathtt{x} := \mathtt{r}_3$
$\mathtt{y} := 2$		

Figure 49: Promising allows the $\mathtt{r}_1 = 2$, despite the acquire load of $\mathtt{y}$.

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
LB	✔	✘	✔	✔	✔	0.03
LBd	✔	✔	✔	✔	✔	0.03
LBfd	✔	✘	✔	✘	✔	0.04
SB	✔	✔	✔	✔	✔	0.11
COH	✘	✔	✔	✔	✔	0.22
2+2W	✔	✔	✔	✔	✔	0.14
ARM-weak	✔	✔	✔	✔	✔	0.05
Par-Inc	✔	✔	✔	✔	✔	0.13
Upd-Stuck	✔	✘	✔	✔	✔	409.99
MP+fences	-	✔	✔	✔	✔	0.04
LBa	✘	✘	✘	✘	✘	0.13
LBa'	✔	✔	✔	✔	✔	0.46
SB+fences	-	✔	✔	✔	✔	0.17

For test LBa, we do not allow the optimisation permitted by Promising: it is contingent on the de-ordering of an acquire read followed by a write.

In IMM, these events remain ordered, so we cannot admit this execution while being a faithful implementation of IMM or RC11.

Grounding Thin-Air Reads with Event Structures

These tests come from Chakraborty and Vafeiadis (2019).

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
CYC	✔	✔	✔	✔	✔	0.05
LB	✔	✘	✔	✔	✔	0.16
LBfd	✔	✘	✔	✘	✔	0.16
RNG	✔	✔	✔	✔	✔	0.58
Coh	✘	✔	✔	✔	✔	0.21
Coh-CYC	✔	✔	✔	✔	✔	25.89
FADD	✔	✔	✔	✔	✔	0.50
Cwrites	✔	✘	✔	✔	✔	259.99

RC11 and IMM both forbid the LBfd (Load Buffering with false dependencies, see **??**) example. This is because of their overly strong restriction of *acyclic*($\sqsubseteq \cup \mathsf{rf}$) which both Grounding and MRD seek to avoid.

The Cwrites example is an outlier on execution time, at 259.99s. This, as with other tests, is because of a large number of sequenced reads and a larger value set ($\mathcal{V} = 0, 1, 2$) required to capture the meaning of the litmus test. A limitation of our approach is that larger programs – particularly programs which need many values to model, or programs with many sequenced reads – generate unwieldy denotations, greatly slowing down the tool. In practice we find our tool to have acceptable execution time for all litmus tests that we have considered.

A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-Air Executions

These tests come from Pichon-Pharabod and Sewell (2016).

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
LBfd	✔	✘	✔	✘	✔	0.16
LB+dep+dep	✔	✔	✔	✔	✔	0.05
LB	✔	✔	✔	✔	✔	0.14

The Problem of Programming Language Concurrency Semantics

These tests come from Batty et al. (2015). This paper introduced the thin-air problem and highlighted how axiomatic approaches to solve the thin air problem cannot work. Indeed, as seen in the table below, both RC11 and IMM do not agree with the literature on the outcomes for tests LB+ctrldata+po or LB+ctrldata+ctrl-double. These programs have false dependencies that a compiler could optimise away, and the simple *acyclic*($\mathsf{po} \cup \mathsf{rf}$) approach falls short.

Test name	$[\![P]\!]$	$[\![P]\!]_{\mathsf{RC11}}$	$[\![P]\!]_{\mathsf{MRD+RC11}}$	$[\![P]\!]_{\mathsf{IMM}}$	$[\![P]\!]_{\mathsf{MRD+IMM}}$	Total time (s)
SB	✔	✔	✔	✔	✔	0.11
MP	✔	✔	✔	✔	✔	0.12
LB	✔	✘	✔	✔	✔	0.45
LB+datas	✔	✔	✔	✔	✔	0.14
LB+ctrldata+po	✔	✘	✔	✔	✔	0.15
LB+ctrldata+ctrl-double	✔	✘	✔	✘	✔	0.15
LB+ctrldata+ctrl-single	✔	✔	✔	✔	✔	0.05
CSE	✔	✘	✔	✘	✔	0.35
RRE	✔	✘	✔	✘	✔	6.85

MRD gives the correct determination for each of these programs, and allows the aggressive optimisations that a language model should permit.

7.3 Web tool

We have adapted MRDer to integrate with a web server, so users can quickly and interactively explore the executions of litmus tests allowed under our memory model. It is designed to be familiar to CPPMem (Batty et al. 2011) users, and presents a list of executions that are permitted by the memory model. The key difference for a user of CPPMem is the inclusion of the sdep relation and our axiom which forbids cycles in $(\mathsf{sdep} \cup \mathsf{rf})$.

The tool is available here: `https://cs.kent.ac.uk/projects/MRDer` and a screenshot is presented in Figure 50.

7.4 Conclusion

MRDer is the first tool to automatically evaluate litmus tests for a fully featured memory model. It was critical in allowing us to validate our semantics as we built it, and we believe that being able to present tables of results for MRD and its augmentations is a useful resource for those wishing to understand our memory model. The results from MRDer have been presented to the SG1 sub-group of the International Standards Organisation (ISO) whose remit includes the specification of the C++ concurrency model. They have described MRD as the "best path forwards" for the future of the concurrency semantics of C++ (Giroux 2019). Tools have been historically been used in memory model construction to great effect (Alglave, Maranget and Tautschnig 2014; Torlak, Vaziri and Dolby 2010; Wickerson et al. 2017), but regrettably they

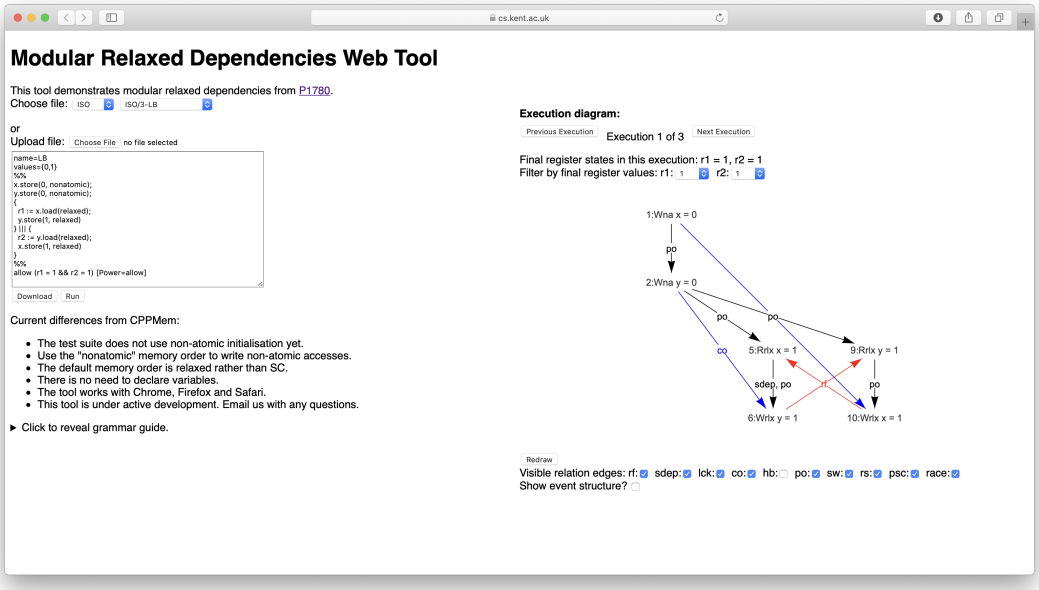


Figure 50: A screenshot of MRDer-web.

had taken a back seat in academic attempts to solve the thin-air problem. It is our hope that the value of tools will once again be recognised and focussed on in future efforts to refine a solution to the thin-air problem, just as their value continues to be recognised industrially.

Chapter 8

Conclusion

In this thesis we have seen how building memory consistency models to avoid the thin air problem has evolved over years of development. As models have more precisely described their hardware and software platforms they have in turn become far more complex and inscrutable. Reasoning about these models, and validating that changes are not introducing new behaviours cannot be left to intuition or approximation. Tools are a critical component of memory model design and validation.

PTX. In Chapter 3 we saw a complex axiomatic memory model extended to incorporate finer architectural detail. These changes were validated through a combination of automated testing, generation of litmus tests, and direct comparison of the formal models – both bounded using Alloy, and unbounded using Coq. At each of the stages, Alloy was the tool used to help make sense of the model. The Alloy artefact enabled comparison between the original model and our extended model, allowing us to generate the litmus tests, and generation of proof burdens Coq which we used to prove that our modifications to the model were a strict extension. In this approach the tool building is directly tied to the formal definition of the model.

PrideMM. In Chapter 4 we looked at a highly complex thin-air free memory model, for which there was no tool. The model is detailed, and while a high-level abstraction of the model is simple to understand, applying the rules directly by hand is fiddly and error prone. We built a tool, PrideMM, in which we could express the mathematical definition of the model in order to execute it automatically. Our tool uses a new breed of solver technology which is well suited to the quantification alternation present in the J+R memory model, and while it does not excel in terms of performance compared to Herd7 or Alloy for traditional axiomatic memory models, it

does provide new expressiveness: allowing us to solve a problem which cannot be written for Herd or Alloy. PrideMM allowed us to input a variety of litmus tests with it, we can replicate the results of Jeffrey and Riely, and in one instance it uncovered a minor mistake of the original authors – even where they had done mechanical proofs of their model using Agda. PrideMM was the first tool built to automatically evaluate a thin-air free memory model over a litmus test.

MRD. In Chapter 5 we introduce a new approach to defining semantic dependency, where calculation of a relation for use in an axiomatic memory model looks at all the possible executions of the program to determine if some outcomes are guaranteed, and that a dependency might be elided. Subsequently, we extend the C++ memory model with this definition of sdep in Chapter 6, and build an automated tool for running litmus tests in Chapter 7. This is only the *second* thin-air free memory model which has an automated testing tool, and gives us confidence that our work can be applied to solve the thin-air problem in the the ISO C++ standard. MRDer is now available online for demonstration in a similar style to `cppmem`.

8.1 A community effort

Solving the OOTA problem requires a semantics which explores all many executions simultaneously, rather than one at a time as axiomatic memory models. While we believe that MRD is the best looking solution to the thin-air problem in C++ its high-level resemblance to the alternatives is important and striking. There is a consensus forming around event structures carrying the necessary information to build a thin-air free model, with Jeffrey and Riely (2016); Pichon-Pharabod and Sewell (2016); Chakraborty and Vafeiadis (2019) all using event structures in their semantics. An outlier is the Promising Semantics which uses an operational model and promises to do a large state-space exploration about future program states, which are used to validate reads.

8.2 Future Work

While MRD+RC11 does solve the thin-air problem for C++, there are still challenges to overcome in the C++ memory model.

8.2.1 C++ Standard Wording

An early draft of standards text exists for our MRD+RC11 model of C++ concurrency, but more work has to happen to get this into the standard. While it is easy to incorporate the semantic dependency relation, incorporating the *calculation* of such a relation will be more difficult. We are working with SG1, the sub-group of the C++ standards committee to get to standards wording which can be accepted into a future standard. Interfacing the mathematical formalisms with the English-language standards text requires careful iteration between us as developers of the formal model, and the ISO as guardians of the standard.

8.2.2 Pointers in C++

So far memory models do not have a comprehensive treatment of pointers, and do not really capture the rich optimisations which compilers apply to pointers. Compilers are able to construct invariants about the object model, such as alignment, which can be done transparently to the programmer. For example Figure 51 presents a C++ litmus test which resulted from discussion with Goldblatt (2019). It us to consider a load buffering test where the compiler uses over alignment of objects to remove dependencies.

```
struct S {
  float vectorData[8];
};

S s0, s1, s2;
S *r1, *r2;

atomic<S*> x{&s0};
atomic<S*> y{&s1};

void t1() {
  r1 = x.load(rlx);
  if ((uintptr_t)r1 % 32 == 0) {
    y.store(&s2);
  }
}

void t2() {
  r2 = y.load(rlx);
  if ((uintptr_t)r2 % 32 == 0) {
    x.store(&s2);
  }
}
```

Figure 51: An example from David Goldblatt, is it possible to observe `r1 == &x && r2 == &y`?

In this program, threads `t1` and `t2` do a dependent store of an address if they observe that the value loaded from r2 is 32-bit aligned. A compiler can invent an invariant that `x` and `y` are always 32 bit aligned, thus eliminating the conditional and making this program very similar to the first LB example in Figure 2. These questions about compiler optimisations on address space remain open.

8.2.3 Global Optimisations and Java

Java's memory model is defined in an operational style, which is not comparable to the axiomatic style of C/C++. MRD satisfies most of the Java Causality Test Cases, leading us to believe that it could make a suitable candidate for a thin-air free Java Memory Model too. There are outstanding questions about Java's handling of thread-inlining optimisations which we have not considered in our simple value oracle for reads in the MRD model. Further, as Java is a just-in-time compiled language, the runtime can gather facts about execution which are unavailable to a statically compiled programming language like C++, or a static interpretation of code such as MRD. JIT enables new types of global optimisations which MRD does not support but we believe it could be adapted. Consideration of JIT and thread-inlining optimisations could lead us to a very usable and high fidelity model for Java.

8.3 Parting thoughts

We now have a solution to the Out-Of-Thin-Air Problem which captures exactly which dependencies are preserved by compilation, and in doing so we have built a memory model which gels neatly with the existing state-of-the-art in the C++ definition. Building this memory model would not have been possible without back and forth between mathematical formalism and tool building to test and refine our ideas. These tools live on past the development stage of the model into the education phase. It is our hope that in the future MRDer is used online as an oracle to programmers for the C++ memory model in just the same way that `cppmem` is used today.

Bibliography

Abdulla, P. A. et al. (2020). The decidability of verification under promising 2.0. 2007.09944.

Alglave, J. and Cousot, P. (2016). Syntax and analytic semantics of LISA. Available at https://arxiv.org/abs/1608.06583.

Alglave, J., Cousot, P. and Maranget, L. (2016). Syntax and analytic semantics of the weak consistency model specification language CAT. Available at https://arxiv.org/abs/1608.07531.

Alglave, J., Maranget, L. and Tautschnig, M. (2014). Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans Program Lang Syst*, 36(2), pp. 7:1–7:74.

Alglave, J. et al. (2011). Litmus: Running tests against hardware. In P. A. Abdulla and K. R. M. Leino, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 41–44.

Apple Inc. (2021). XNU Kernel. Available at https://opensource.apple.com/source/xnu/xnu-7195.81.3/.

ARM (2014). *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*.

ARM (2016). *ARM® Compiler armasm User Guide*.

ARM (2020). *Arm Architecture Reference Manual ARMv8*.

Batty, M. (2011). *The C11 and C++11 Concurrency Model*. Ph.D. thesis, University of Cambridge.

Batty, M., Donaldson, A. F. and Wickerson, J. (2016). Overhauling SC atomics in C11 and opencl. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 634–648.

Batty, M. et al. (2011). Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 55–66.

Batty, M. et al. (2015). The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 283–307.

Biere, A., Lonsing, F. and Seidl, M. (2011). Blocked clause elimination for QBF. In *The 23rd International Conference on Automated Deduction CADE*.

Blanchette, J. C. and Nipkow, T. (2010). Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, eds., *Interactive Theorem Proving*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 131–146.

Bornholt, J. and Torlak, E. (2017a). Ocelot: A solver-aided relational logic DSL.

Bornholt, J. and Torlak, E. (2017b). Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pp. 467–481.

Bove, A., Dybjer, P. and Norell, U. (2009). A brief overview of Agda - A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban and M. Wenzel, eds., *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, *Lecture Notes in Computer Science*, vol. 5674, Springer, pp. 73–78.

Chakraborty, S. and Vafeiadis, V. (2019). Grounding thin-air reads with event structures. *PACMPL*, 3(POPL), pp. 70:1–70:28.

Claessen, K. and Sörensson, N. (2003). New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*.

Deacon, W. and Alglave, J. (2019). The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat.

Flur, S. et al. (2017). Mixed-size concurrency: ARM, POWER, C/C++11, and SC. *SIGPLAN Not*, 52(1), p. 429–442.

Foundation, H. (2015). HSA Platform System Architecture Specification. Available at `http://www.hsafoundation.com/html/Content/PRM/Topics/06_Memory/memory_model.htm`.

Giroux, O. (2019). ISO WG21 SG1 concurrency subgroup vote, unanimously approved: OOTA is a major problem for C++, modular relaxed dependencies is the best path forward we have seen, and we wish to continue to pursue this direction.

Goldblatt, D. (2019). private correspondence.

Gray, K. E. et al. (2015). An integrated concurrency and core-isa architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pp. 635–646.

Group, C. S. R. (1983). BSD Unix version 4.2. Available at `https://www.tuhs.org/cgi-bin/utree.pl?file=4.2BSD`.

Intel Corporation (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

ISO/IEC (2010). *Programming languages – C++*. Draft N3092.

Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Trans Softw Eng Methodol*, 11(2), pp. 256–290.

Jagadeesan, R., Jeffrey, A. and Riely, J. (2020). Pomsets with preconditions: A simple model of relaxed memory. *Proc ACM Program Lang*, 4(OOPSLA), pp. 1–30.

Janota, M. (2018). Towards generalization in QBF solving via machine learning. In *AAAI Conference on Artificial Intelligence*.

Janota, M., Grigore, R. and Manquinho, V. M. (2017). On the quest for an acyclic graph. *CoRR*, abs/1708.01745, `1708.01745`.

Janota, M. et al. (2016). Solving QBF with counterexample guided refinement. *Artificial Intelligence*, 234, pp. 1–25.

Jeffrey, A. and Riely, J. (2016). On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, New York, NY, USA: ACM, LICS '16, pp. 759–767.

Jordan, C., Klieber, W. and Seidl, M. (2016). Non-CNF QBF solving with QCIR. In *AAAI Workshop: Beyond NP*, *AAAI Workshops*, vol. WS-16-05, AAAI Press.

Kang, J. et al. (2017). A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 175–189.

Khronos Group (2020). Vulkan® 1.1.164 - A Specification.

Lahav, O., Giannarakis, N. and Vafeiadis, V. (2016). Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 649–662.

Lahav, O. et al. (2017). Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pp. 618–632.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans Computers*, 28(9), pp. 690–691.

Lewis, H. R. (1980). Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3), pp. 317–353.

Libkin, L. (2004). *Elements of Finite Model Theory*. Springer.

Lochbihler, A. (2014). Making the Java memory model safe. *ACM Trans Program Lang Syst*, 35(4).

Lustig, D. et al. (2017). Automated synthesis of comprehensive memory model litmus test suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: ACM, ASPLOS '17, pp. 661–675.

Manson, J., Pugh, W. and Adve, S. V. (2005). The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pp. 378–391.

Maranget, L., Sarkar, S. and Sewell, P. (2012). A tutorial introduction to the ARM and POWER relaxed memory models.

McKenney, P. E. et al. (2016). Out-of-thin-air execution is vacuous. *ISO/IEC JTC1 SC22 WG21*, P0422(R0).

Milicevic, A. et al. (2015). Alloy*: A general-purpose higher-order relational constraint solver. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 609–619.

Moiseenko, E. et al. (2019). Reconciling event structures with modern multiprocessors. `1911.06567`.

Mozilla Project (2021). Firefox. Available at `https://hg.mozilla.org/mozilla-central/`.

NVIDIA (2014). Adaptive Parallel Computation with CUDA Dynamic Parallelism. Available at `https://https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/`.

NVIDIA (2018). Programming Guide :: CUDA Toolkit Documentation. Available at `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

NVIDIA Corporation (2019). Parallel thread execution ISA version 6.5. Available at `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

Ou, P. and Demsky, B. (2018). Towards understanding the costs of avoiding out-of-thin-air results. *Proc ACM Program Lang*, 2(OOPSLA), pp. 136:1–136:29.

Paviotti, M. et al. (2020). Modular relaxed dependencies in weak memory concurrency. In P. Müller, ed., *Programming Languages and Systems*, Cham: Springer International Publishing, pp. 599–625.

Pichon-Pharabod, J. and Sewell, P. (2016). A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 622–633.

Podkopaev, A., Lahav, O. and Vafeiadis, V. (2019). Bridging the gap between programming languages and hardware weak memory models. *Proc ACM Program Lang*, 3(POPL).

Project, T. C. (2021). Chromium. Available at `https://chromium.googlesource.com/chromium/src`.

Pugh, W. (1999). Fixing the java memory model. In *Proceedings of the ACM 1999 Conference on Java Grande*, New York, NY, USA: Association for Computing Machinery, JAVA '99, p. 89–98.

Pugh, W. (2004). Java causality test cases. Available at `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`.

Pulte, C. et al. (2018). Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. *PACMPL*, 2(POPL), pp. 19:1–19:29.

QBFLIB (2017). QBF Eval 2017. Available online at `http://www.qbflib.org/event_page.php?year=2017`.

Reger, G., Suda, M. and Voronkov, A. (2016). Finding finite models in multi-sorted first-order logic. In N. Creignou and D. L. Berre, eds., *Theory and Applications of Satisfiability Testing - SAT*, *Lecture Notes in Computer Science*, vol. 9710, Springer, pp. 323–341.

Reynolds, A. et al. (2013a). Finite model finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pp. 640–655.

Reynolds, A. et al. (2013b). Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, ed., *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, *Lecture Notes in Computer Science*, vol. 7898, Springer, pp. 377–391.

Reynolds, A. et al. (2016). Model finding for recursive functions in SMT. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pp. 133–151.

Riely, J., Jagadeesan, R. and Jeffrey, A. (2020). private correspondence.

Sarkar, S. et al. (2011). Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pp. 175–186.

Sewell, P. et al. (2010). X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun ACM*, 53(7), p. 89–97.

The Coq Development Team (2017). Coq. Available at `https://coq.inria.fr`.

Torlak, E. and Bodik, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, PLDI '14, pp. 530–541.

Torlak, E. and Jackson, D. (2007). Kodkod: A relational model finder. In O. Grumberg and M. Huth, eds., *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on*

Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4424, Springer, pp. 632–647.

Torlak, E., Vaziri, M. and Dolby, J. (2010). MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, PLDI '10, pp. 341–350.

Torvalds, L. (2021). The Linux kernel. Available at `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree`.

Wickerson, J. et al. (2017). Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 190–204.

Winskel, G. (1987). *Event structures*, Berlin, Heidelberg: Springer Berlin Heidelberg, chap. 8. pp. 325–392.

Zhang, J. and Zhang, H. (1995). SEM: a system for enumerating models. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI*, Morgan Kaufmann, pp. 298–303.