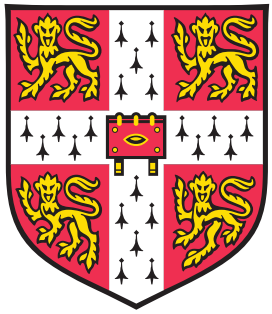
A no-thin-air memory model for programming languages

Jean Yves Alexis Pichon-Pharabod
Trinity Hall
University of Cambridge



September 2017

This dissertation is submitted for the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text.

This thesis does not exceed 60,000 words.

The introduction is based on Pichon-Pharabod and Sewell [65] and Batty et al. [12]. Chapters 3 and 4 are based on Pichon-Pharabod and Sewell [65]. Chapters 6, 7, 8 and 9 are partly based on it too. The parts of Chapter 8 about testing and common subexpression elimination are based on Frida J. Tveit's Part II project. Examples from Chapter 6 are taken from various sources, as described there. Chapter 11 is based on Pichon-Pharabod and Sewell [65], and discusses other people's work.

Abstract

Many hardware and compiler optimisations introduced to speed up single-threaded programs also introduce additional, sometimes surprising, behaviours for concurrent programs with shared mutable state. How many of these extra behaviours occur in practice depends on the combination of the hardware, compiler, runtime, etc. that make up the platform. A memory model, which prescribes what values each read of a concurrent program can read, allows programmers to determine whether a program behaves as expected without having to worry about the details of the platform. However, capturing these behaviours in a memory model without also including undesirable "out-of-thin-air" behaviours that do not occur in practice has proved elusive. The memory model of C and C++ allows out-of-thin-air behaviour, while the Java memory model fails to capture some behaviours that are introduced in practice by compiler optimisations.

In this thesis, we propose a memory model that forbids out-of-thin-air behaviour, yet allows the behaviours that do occur. Our memory model follows operational intuitions of how the hardware and compilers operate. We illustrate that it behaves as desired on a series of litmus tests. We show that it captures at least some of the expected behaviours, that it forms an envelope around some common compiler optimisations, and that it is implementable on common hardware using the expected compilation schemes. We also show that it supports some established programming idioms.

Acknowledgements

My supervisor, Peter Sewell. My examiners, Neel Krishnaswami and Viktor Vafeiadis. My assessors, Alan Mycroft and Andy Pitts. My coauthors, Mark Batty, Lars Birkedal, Kayvan Memarian, Kyndylan Nienhuis, Filip Sieczkowski, Kasper Svendsen, and Frida Tveit. For interesting discussions and remarks, Richard Bornat, Simon Castellan, Ohad Kammar, Robin Morrisset, and Jaroslav Ševčík. My office mates, Shaked Flur and Hannes Mehnert. The PLS group at the Computer Laboratory. For the vector file of the crest of Trinity Hall, Kayvan Memarian. For encouraging me to apply to Cambridge, James Leifer. The administrative staff of the Computer Laboratory, especially Lise Gough. My friends, and my family.

À M^{me} W.

Contents

Notation

tt denotes the boolean "true"

ff denotes the boolean "false"

$[]$ denotes the empty list

$\ell_1 +\!\!+ \ell_2$ denotes the concatenation of the lists ℓ_1 and ℓ_2

$e \# e'$ denotes conflict, see Page 54

$e \sim e'$ denotes immediate conflict, see Page 54

$\rightsquigarrow$ denotes steps of the semantics

`;` is the statement separator, and also denotes concatenation of statements

$\{a \mapsto b\} = \{(a, b)\}$

$f[a \mapsto b] = \{(a', b') \mid (a', b') \in f \wedge a' \neq a\} \cup \{(a, b)\}$

$\mathrm{fst}((a, b)) = a$

$\mathrm{snd}((a, b)) = b$

$A \uplus B = A \cup B,$ but is only defined when $A \cap B = \varnothing$

The syntax of the language we use for our examples is described on Page 21.

In a litmus test postcondition, `x = v` means that the last write to `x` in coherence order is a write of v.

Chapter 1

Introduction

1.1 Context

Multiprocessors appeared as early as 1962, with the Burroughs D-825 [30], but remained rare for many years, as it was often simpler, cheaper, and more usable to increase CPU clock frequency and exploit instruction-level parallelism. However, in the early 2000s, hardware designers found themselves too constrained by the challenges of power dissipation to continue increasing frequency, and unable to extract any more instruction-level parallelism. Instead, they had to resort to designing multiprocessors, in the form of multicore processors, that is, multiprocessors on a single piece of silicon [37]. However, as for any multiprocessor, to actually take advantage of the joint computing power of the cores, many tasks require the cores to communicate. In mainstream multicore processors, this communication is done via a shared, mutable memory through which cores communicate by writing and reading. This memory is implemented by a series of buffers and caches on top of a flat memory that propagate writes and reads from one core to the others. However, this mechanism is typically not transparent: a program executing on multiple cores can observe that there is something interposed between its threads, and that memory is not *sequentially consistent* [45], that is, an immediately updated map from locations to values. This appeared as early as 1972 with the IBM 370/158MP [12], but became widespread in the 2000s with multicore processors. Moreover, on many architectures, cores have instruction pipelines that execute independent instructions, for example writes to different memory locations, out of order, in a way that programs can observe. This compounds with the effect of caches and buffers. This setting where memory is not sequentially consistent is called *relaxed*, or *weak*.

As the exact extent of the relaxed behaviours depends on details of the hardware configuration, programmers and hardware designers benefit from a *memory model* of the hardware, that is, a description of the allowed behaviours. The memory model of the hardware acts as a contract between the programmer and hardware designers. Programmers can work against the memory model rather than the details of the hardware, and hardware designers can change the details of the hardware, as long as they follow the memory model. The design of the memory model is in tension between two goals. (1) The memory model needs to allow enough behaviours to be implementable efficiently. (2) At the same time, the memory model needs to forbid enough behaviours to be programmable efficiently. Programmers should not need to introduce so many synchronisation instructions to ensure the outcome of their programs that they cancel the benefits of having a relaxed memory in the first place. Moreover, the memory model benefits from being

abstract, so that programmers do not need to know about the details of the hardware, and so that hardware designers can change those details.

Programming languages add their own challenge to relaxed memory. While a machine-code program is meant to be taken at face value by the hardware, a program in a programming language alludes to some machine-code program that — one hopes — executes fast. Compilers are expected to deliver such a machine-code program. To do so, they are allowed to take certain liberties with the program text concerning how the flow of the program is expressed, the pattern of accesses to the memory, etc. The extent of these liberties was not designed, but rather accumulated over time, from the early work of Allen [8] to sparse conditional constant propagation [88] and beyond. These liberties allow a compiler to optimise a single-threaded program, that is, replace it with a program the behaviours of which are behaviours of the original, but is — one hopes — faster. However, when applied to a multi-threaded program, these 'optimisations' can introduce extra behaviours, making the term misleading. By the time the extent of this interaction was realised, for example by Boehm [19], programmers were unwilling to give up on established compiler optimisations, and the effect of the state of the art in implemented compiler optimisations was restricted enough for programming with the extra behaviours to still be possible in practice. However, describing the extent of the extra behaviours, that is, designing a memory model for a programming language, is an open problem. Batty et al. [12] note that:

> Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still does not have a credible proposal for the concurrency semantics of any general-purpose high-level language that includes high-performance shared-memory concurrency primitives. This is a major open problem for programming language semantics.

Neither of the main previous attempts succeeds in this. The Java Memory Model [49] is unsound with respect to standard compiler optimisations [26, 75]. The memory model of C11 and C++11 (abbreviated C/C++11) [20, 18, 2, 15] is arguably the current state of the art, and gives the "right" behaviours in many cases, but it permits too much in others. At the heart of the problem are the "thin-air" examples for concurrent high-performance accesses, recalled in Section 1.2.1, in which values appear out of nowhere [49, 75, 21, 12]. Those thin-air executions are not thought to occur in practice, with any combination of current compiler and hardware optimisations, but excluding them without also excluding important optimisations has not previously been achieved. A further concern for C/C++11, also highlighted by Batty et al. [12], is the interaction between undefined behaviour and relaxed memory, which we recall in Section 1.2.4.

1.2 Recalling the problems

1.2.1 The thin-air problem

As acknowledged by the C++11 standard [18], in trying to define an envelope around all reasonable optimisations, the C/C++11 memory model also admits undesirable executions where values seem to appear out of thin air. In Section 23.9, paragraph 9, the C++11 standard says (in C++11 syntax):

> [Note: The requirements do allow `r1 == r2 == 42` in the following example, with `x` and `y` initially zero:

```
// Thread 1:
r1 = x.load(memory_order_relaxed);
if (r1 == 42) y.store(r1, memory_order_relaxed);
// Thread 2:
r2 = y.load(memory_order_relaxed);
if (r2 == 42) x.store(42, memory_order_relaxed);
```

However, implementations should not allow such behavior. – end note]

In this thesis, we use a more compact syntax, detailed in Section 2.1, in which this *litmus test* is as follows:

LB+ctrldata+ctrl-single

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	if (r2 == 42)
store$_{rlx}$(y,r1)	store$_{rlx}$(x,42)
r1 = r2 = 42 BAD	

The program starts with two locations `x` and `y`, initially set to 0.
It has two threads:

1. The first thread reads from location `x` into `r1` with a *relaxed* read. A relaxed read or write in C/C++ is meant to be compiled to a plain load or store, without memory barriers or other synchronisation. If this read reads 42, then the first thread writes what it read (the contents of `r1`, which was checked by the test to be 42) to `y` with a relaxed write.

2. The second thread almost does the symmetric of the first thread: it reads from `y` into `r2` with a relaxed read. If it read 42, then it writes 42 to `x` with a relaxed write.

We do not want this program to be allowed to end with `r1` and `r2` both being set to 42. For this outcome to happen, the read of the first thread would have to read 42, which it must read from the write of 42 of the second thread, as there are no other writes to `x`, and `x` is initially 0. The write of 42 of the second thread is only executed if the read of the second thread reads 42, which it must read from the write of the first thread (as there are no other writes to `y`, and `y` is initially 0). For the write of the first thread to write 42, the first thread must have read 42. This means that for the read of the first thread to read 42, it must read it from a write that itself depends on the read of the first thread reading 42. The only way this could happen is if one of the reads read the value 42 "out of thin air".

Having such "out of thin air" outcomes would be highly undesirable: not being able to constrain what values reads read breaks elementary programming and reasoning principles [11, 21, 84].

There is however no precise definition of what thin-air behaviour is—if there were, it could simply be forbidden by fiat, and the problem would be solved. Rather, there are a few known litmus tests like the one above where certain outcomes are undesirable and do not appear in practice as the result of hardware and compiler optimisations. The problem is to draw a fine line between those undesirable outcomes and the outcomes of other very similar litmus tests which important optimisations make observable and which therefore must be admitted.

Java Similar issues arose in the first Java Memory Model design, in a slightly different form [49]. C/C++11 has nonatomic and relaxed accesses, both of which are intended to be implementable without memory barriers or other synchronisation; concurrent relaxed access to the same location is permitted, while concurrent nonatomic access gives wholly undefined behaviour (to let optimisers assume the latter does not occur). Java plain (non-volatile) accesses are similarly intended to be implementable with just the underlying hardware plain loads and stores; programmers are not supposed to make non-synchronised concurrent use of these; however, to provide safety guarantees even in the presence of arbitrary code, the semantics must forbid the forging of pointers. However, if reads are allowed to read values "out of thin air", then such guarantees cannot be provided.

1.2.2 Per-candidate-execution semantics does not suffice

A common approach to relaxed-memory semantics, and that followed by C/C++11, is to define what is called an *axiomatic* memory model. One defines a notion of candidate execution, each consisting of a set of memory actions and basic relations over them (such as program order, reads-from, coherence, etc., see below), and a consistency predicate that picks out the candidate executions that are allowed by the semantics (e.g. including a check that some happens-before relation, derived from the basic relations, is acyclic). The semantics of a program is taken to be the set of all consistent executions that are compatible with some control-flow unfolding of the program, or, in some semantics, that set modulo the existence of data races.

However, as observed by Batty et al. [12], there are pairs of programs which share a particular candidate execution where that execution should be allowed for one program but not for the other. Consider the following:

LB+ctrldata+ctrl-double

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	if (r2 == 42)
store$_{rlx}$(y,r1)	store$_{rlx}$(x,42)
	else
	store$_{rlx}$(x,42)

As there is a write to x in both branches of the second thread of LB+ctrldata+ctrl-double, the conditional can be collapsed by compiler optimisation to yield

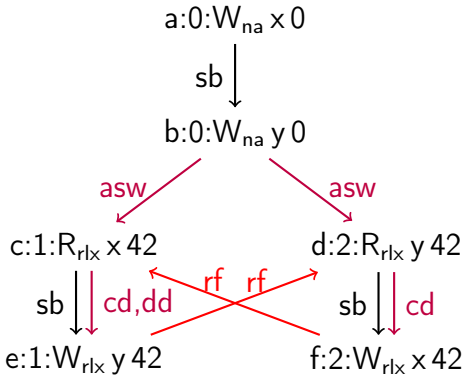
LB+ctrldata+po

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	store$_{rlx}$(x,42)
store$_{rlx}$(y,r1)	

in which both threads can read 42, because the read and the write of the second thread can be reordered, either thread-locally (by the compiler or the hardware) or when propagating between the threads (by the hardware). This happens in practice, and so should be allowed by the semantics.

On the other hand, for LB+ctrldata+ctrl-single (the first example, from the C++ standard), this outcome does not appear in practice; that would, as discussed earlier, be highly undesirable.

In the C/C++11 semantics (detailed in Section 2.3), the two share the candidate execution below, where "sb" ("sequenced-before") is program order, that is, the syntactic order coming from the program text, "asw" indicates thread synchronisation (here from the initial thread to spawned threads), and "rf" indicates where the reads read from. The other edges indicate syntactic dependencies: "dd" stands for (syntactic) data dependency, and "cd", which is not part of C/C++11, stands for (syntactic) control dependency.



Dependencies As the example above shows, syntactic data and control dependencies do not provide enough information to draw the distinction between desirable and undesirable outcomes. In retrospect, this is not surprising: programmers expect to be able to interchange control flow and dependencies without changing program outcome, and compilers do that freely, sometimes removing syntactic dependencies.

For example, LB+ctrl+po below ought to behave the same as LB+ctrldata+po, as the data dependency in LB+ctrldata+po does not contribute any information that the control dependency does not already contribute: the write to `y` can only be a write of 42, and compilers will routinely transform LB+ctrldata+po into LB+ctrl+po.

LB+ctrl+po

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	store$_{rlx}$(x,42)
store$_{rlx}$(y,42)	

Perspective Trying to draw a more general, informal conclusion, Batty et al. [12, §6] argue that the problem with the C/C++11 memory model is that it considers executions individually, whereas what it is trying to model the effect of, the hardware and compiler optimisations, effectively consider all executions simultaneously by operating on the code. For example, executing the write in the second thread of LB+ctrldata+po before the read that is before it in program order is justified because the write occurs in all control-flow paths. Similarly, optimising LB+ctrldata+ctrl-double into LB+ctrldata+po is justified because the write of the second thread occurs in all control-flow paths.

1.2.3 Further challenges

Merging Compiler optimisations can also merge memory actions. For example, for the nonatomic version of the below, compiler passes like common subexpression elimination can and do merge the reads of `y`. The same optimisation for relaxed atomics, turning it

into the second thread of LB+ctrldata+ctrl-double, has been proposed [10]. This would allow the outcome where `r1` and `r2` are set to 42.

JMM CSE trap

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	if (r2 == 42) {
store$_{rlx}$(y,r1)	r3 = load$_{rlx}$(y);
	store$_{rlx}$(x,r3)
	} else {
	store$_{rlx}$(x,42)
	}

Memory models for programming languages need to account for these optimisations. However, the Java Memory Model fails to give this particular litmus test the right behaviour, as noted by Ševčík and Petri [74] (we describe this in more detail in Section 2.4).

Value-range analysis The optimisations considered so far are thread-local, and independent of the manipulated values, and of the wider program in which a thread executes. However, their applicability can be greatly extended by analyses. For example, a compiler pass could analyse the values that `x` and `y` can take during executions of the program below, a variant of the Causality Test Case 1 of Pugh [68]. This analysis, the kind of which we will refer to as "value-range analysis", could determine that `r2` will always contain a nonnegative value. The compiler could then collapse the `if`, and turn this program into LB+ctrldata+po, and therefore allow the outcome where `r1` and `r2` are set to 42.

Java causality test case 1 variant

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	if (r2 >= 0)
store$_{rlx}$(y,r1)	store$_{rlx}$(x,42)

While we are not aware of this precise optimisation being done, counterparts on pointers are performed by alias analyses [22].

Moreover, analysis during just-in-time compilation can take advantage of information about the current state of the program, for example that a write has propagated to a thread, masking previous values and hence shrinking the set of possible values of a variable, thereby enabling additional optimisations. For example, in the program below, after the write of 1 to `z` by the third thread has propagated to the second thread, the conditional on `r3` in the second thread can be collapsed, yielding LB+ctrldata+ctrl-double, and therefore allowing the outcome where `r1` and `r2` are set to 42. While C/C++11 is not typically compiled just-in-time, the combination of inlining and special-casing has similar effects.

Guarded LB+ctrldata+ctrl-double

x = y = z = 0		
r1 = load$_{rlx}$(x);	r3 = load$_{rlx}$(z);	store$_{rlx}$(z,1)
if (r1 == 42)	r2 = load$_{rlx}$(y);	
store$_{rlx}$(y,r1)	if (r2 == 42)	
	if (r3 == 1)	
	store$_{rlx}$(x,42)	
	else	
	store$_{rlx}$(x,42)	

To prevent undesirable behaviour, and in particular thin-air reads, such analysis has to be stable under the effect of the optimisations it enables. In the program below, without optimisations taking advantage of value-range analysis, the read of the second thread cannot read 42. However, if an optimisation takes advantage of that information to reorder the write of the second thread, then the read of the second thread can read 42. This violates the assumption it made earlier, and makes the program execute a write of 42 to `x` even though it should not execute any if the second thread reads 42.

Java causality test case 1 broken variant

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
if (r1 == 42)	if (r2 < 42)
store$_{rlx}$(y,42)	store$_{rlx}$(x,42)

Importantly, this means that optimisations do not necessarily increase the behaviour of threads. An optimisation based on value-range analysis can allow more the behaviour for one thread, for example allowing it to execute a write early, which can allow less behaviour of another thread, for example forbidding it from assuming that a certain read cannot read a certain value, and therefore forbidding it from executing another write early, because the early write now makes this reading this value possible. This also compounds with the fact that merging can restrict the behaviour of a thread, for example preventing it from reading two different values from the same location and therefore preventing it from executing a write, which can allow another thread to assume that it cannot read a certain value (the one written by the write that is now impossible), and therefore allow that other thread more behaviour. This defeats the hope of identifying a maximally optimised program.

1.2.4 The concurrent undefined behaviour problem

The C/C++11 memory model also has an issue with its treatment of undefined behaviour, that is separate from, but interacts with, the "out-of-thin-air" problem. If a language features undefined behaviour, as C and C++ do, then its memory model needs to be able to express it, to determine whether it is triggered. As pointed out by Batty et al. [12, §7], there is a mismatch in the C/C++ standard: the thread-local semantics, which is described operationally, assumes that there is some form of execution order that makes it possible to tell whether a point of the program has been reached, and therefore undefined behaviour is triggered (though if undefined behaviour *is* triggered, it makes the whole program undefined, not merely the execution from that point). However, the candidate executions of the axiomatic memory model do not have such a notion, and because they are candidate *complete* executions, rather than being built incrementally, it is far from obvious how it could be included. We illustrate this problem and explain how our memory model accounts for it in Section 6.7.

1.3 Thesis

Our thesis is that it is possible to address these problems:

> It is possible to design memory models for programming languages, that are weak enough to be sound with respect to hardware- and compiler-induced

behaviour, but strong enough to be usable, by excluding out-of-thin-air behaviour. Moreover, it is also possible to handle undefined behaviour in concurrent contexts.

Plan We support this thesis by presenting a memory model (Chapters 3, 4, and 5), arguing that it behaves well on a series of litmus tests (Section 6), establishing that it can be implemented using the expected compilation schemes (Chapter 7) and that it forms an envelope around some common compiler optimisations (Chapter 8), and showing that it supports some common programming idioms (Chapter 9). We then discuss the remaining challenges (Chapter 10).

Summary of our approach In our memory model, we represent the state of a thread using an event structure, that is, a set of events equipped with some causal order and a conflict relationship obeying certain sanity conditions. The event structure represents all the possible future executions of the thread. The key idea of our memory model is that a thread can take transitions that mutate its event structure that account for mixed execution and thread-local and inter-thread optimisations. Threads interact by synchronising through a storage subsystem.

Chapter 2

Background

In this chapter, we first introduce the syntax of the calculus we will consider, and discuss general features of memory models. We then describe the two most developed programming language memory models, those of C/C++11 and Java, as they delineate the problem well: the former allows too much behaviour, and the latter too little. Finally, we describe the hardware memory models of Power, ARM, and x86, to provide context and a counterpoint to the challenges of designing a memory model for a programming language, and because our memory model reuses the storage subsystem of Power (as described in Section 4.4.2). We leave discussion of other related work to Chapter 11.

2.1 Language

We consider a minimal calculus featuring the core concurrency primitives of C/C++11, but (as explained below) abstracting over the subtleties of the thread-local semantics. This calculus is inspired by that of the CppMem C/C++11 memory model exploration tool [14]. It features relaxed-atomic, nonatomic, and sequentially consistent reads and writes, acquire reads, release writes, read-modify-writes, locks, and a thread-local undefined behaviour trigger. This covers all the C/C++11 memory orders except "consume".

Definition 1. The syntax of our calculus is defined inductively by

p	::=	program
	$\mid$ ss_1 `||` ... `||` ss_n	parallel threads
s	::=	statement
	$\mid$ `r =` $load_{rmo}$`(x)`	read
	$\mid$ $store_{wmo}$`(x,e)`	write
	$\mid$ `r1, r2 =` $rmw_{acqrel/acq}$`(x,e,e')`	read-modify-write
	$\mid$ `if (`e_1 `cmp` e_2`)` ss_1 `else` ss_2	conditional
	$\mid$ `r = e`	register assignment
	$\mid$ `lock` ℓ	lock
	$\mid$ `unlock` ℓ	unlock
	$\mid$ `undef`	undefined behaviour
ss	::=	statements
	$\mid$ s_1`; ..  ;` s_n	sequential composition
	$\mid$ `{ ss }`	block
e	::=	pure expressions

	$\mid$ `v`	constant
	$\mid$ `r`	register
	$\mid$ $\texttt{e}_1$ `+` $\texttt{e}_2$	sum
	$\mid$ $\texttt{e}_1$ `-` $\texttt{e}_2$	difference
	...	
`cmp`	::=	comparison operator
	$\mid$ `==`	equality
	$\mid$ `!=`	inequality
	$\mid$ `>=`	greater than
	...	
`rmo`	::=	read memory orders
	$\mid$ `rlx`	relaxed
	$\mid$ `acq`	acquire
	$\mid$ `sc`	sequentially consistent
	$\mid$ `na`	nonatomic
`wmo`	::=	write memory orders
	$\mid$ `rlx`	relaxed
	$\mid$ `rel`	release
	$\mid$ `sc`	sequentially consistent
	$\mid$ `na`	nonatomic

Variables `x`, `y`, ..., are shared memory locations, and actions on them induce memory actions, whereas `r1`, `r2`, ..., are registers, thread-local variables which have no effect on memory, and ℓ is a lock location.

The registers in our calculus do not feature in usual programming languages, and are unrelated to the C/C++11 "register" keyword. We introduce them in our calculus for two purposes. First, they are used to explicitly represent data flow, which allows us to have simple, well-separated constructs in our language, instead of the complex compound constructs of real programming languages. For example, `y = x++` in C/C++11 can be expressed (if we focus on data flow and ignore the many important details, for example as developed by Memarian et al. [57]) in our calculus as

```
r1 = load_na(x)
r1 = r1 + 1
store_na(x,r1)
store_na(y,r1)
```

Second, they can be used to elide irrelevant memory actions. For example, `int y = x` in C/C++11, where `y` is purely thread-local, corresponds to

```
r1 = load_na(x)
store_na(y,r1)
```

despite the fact that the store to `y`, or in fact the existence of `y`, is irrelevant to concurrency, and that the program could more simply be written as

```
ry = load_na(x)
```

While most programming languages do not feature such register variables, intermediate languages in compilers, for example registers in LLVM IR [4], or in semantics, for example the CppMem C/C++11 memory model exploration tool [14], or the Core language in the Cerberus C semantics [57], do. Moreover, optimisations like scalar replacement of aggregates transform the program with the irrelevant variable `y` above into the one where it has been replaced by a register. We return to this in Section 8.4 in the context of common subexpression elimination.

Undefined behaviour In C/C++11, in addition to memory errors, undefined behaviour can be induced by some operations, for example division by zero. In our calculus, for clarity, we separate thread-local undefined behaviour from other operations, with an `undef` statement that signals undefined behaviour.

Location typing As in the formalised C/C++ memory model, for simplicity, we assume a location typing: each location can be used for either atomic locations, or nonatomic locations, or locks. We return to this limitation in Section 10.4.

Finite domain We restrict values (`v` above) to a finite domain V. Our calculus does not feature pointers, or different types of values, as they are not directly relevant to out-of-thin-air. While the out-of-thin-air problem is independent from pointers, it is especially problematic in the presence of pointers, as it violates physical separation, as illustrated by "ghostly linking of data structures" example of Boehm and Demsky [21]. We return to this limitation in Section 10.1.

Address dependencies Our language does not feature computed addresses, or address dependencies. We return to this limitation in Section 10.3.

Loops Our language omits loops. This is not for any fundamental reason, but rather to keep the semantics straightforwardly executable. We return to this limitation in Section 10.2.

Thread composition Our language features a top-level concurrent composition of threads (the "parallel" operator `||`) rather than the thread management mechanism of C/C++11. This allows us to consider litmus tests without clutter, at the cost of limiting us to a bounded number of threads.

RMWs Our language features a read-modify-write (abbreviated RMW, and also called 'atomic update') operation, `rmw`. Given a location, a value to be expected, and a replacement value, it tries to atomically read the expected value from the location, and replace it with the replacement value, returning a success/failure flag, and the value read. This specific kind of RMW is also called 'compare-and-swap' or 'compare and exchange'. The RMW we model reads with the same strength as an acquire read, and writes with the same strength as a release write. We return to this limitation in Section 10.5.

2.2 Basic memory model concepts

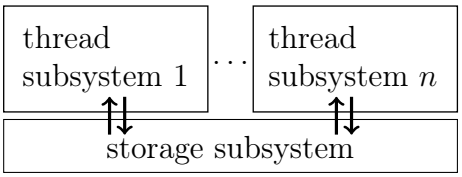


Figure 2.1: Schema of a memory model split into a storage subsystem and thread subsystems

Styles There are several styles of memory models: the *operational* style, where the memory model is defined by the steps of an abstract machine, as in our case; the *axiomatic* style[1], where a complex, global predicate is used to filter a coarse set of potential executions of a program (the C/C++11 memory model, sketched in Section 1.2.2, is an example of this style); the *denotational* style, where the description of a program is built inductively. Because of the problem they try to address, many memory models are made from a combination of different styles.

Thread subsystem and storage subsystem To simplify presentation, and to reflect the architecture of the underlying hardware, memory models, in particular operational memory models, are often split into two parts, as sketched in Figure 2.1: a *storage subsystem*, through which threads communicate, that describes the propagation effects between threads, and *thread subsystem*s, that describe the out-of-order execution of instructions by the threads.

Coherence An important feature of most memory models (C/C++11, Power, ARM, x86, but *not* Java, see Causality Test 16 on Page 83) is *coherence*. Coherence is the existence, for each location, of a total order on the writes at that location that respects program order, and that reads respect. C/C++11 expresses coherence using a "modification order" relation on writes. The following litmus test illustrates one aspect of coherence, "read-read coherence":

CoRR

x = 0	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$	$\texttt{r1 = load}_{\texttt{rlx}}\texttt{(x);}$
$\texttt{store}_{\texttt{rlx}}\texttt{(x,2)}$	$\texttt{r2 = load}_{\texttt{rlx}}\texttt{(x)}$
$\texttt{r1 = 2} \wedge \texttt{r2 = 1}$ BAD	

Coherence respects program order, so the write of 1 is before the write of 2 in coherence order. Reads from the same thread respect coherence, so if the first read reads from the write of 2, the second read cannot read from the write of 1. This is not enforced if the locations are different, as illustrated by the MP litmus test on Page 86.

The following litmus illustrates that what is respected is coherence, not merely program order of writes: if the third thread observes that the write of 1 is before the write of 2 in coherence order, by reading 1, then 2, then the fourth thread cannot read 2, then 1, as that would violate the coherence order we know from the third thread.

[1]This is different from the usual meaning of "axiomatic semantics", where the semantics of a programming language is given by a program logic.

CoRR2

x = 0			
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(x,2)}$	$\texttt{r1 = load}_{\texttt{rlx}}\texttt{(x);}$ $\texttt{r2 = load}_{\texttt{rlx}}\texttt{(x)}$	$\texttt{r3 = load}_{\texttt{rlx}}\texttt{(x);}$ $\texttt{r4 = load}_{\texttt{rlx}}\texttt{(x)}$
$\texttt{r1 = 2} \wedge \texttt{r2 = 1} \wedge \texttt{r3 = 1} \wedge \texttt{r4 = 2}$ BAD			

Non-multi-copy-atomicity Writes at different locations propagate to each thread individually. This important feature of many relaxed memory models (in particular Power and ARMv7) is called non-multi-copy-atomicity in the terminology of Collier [28]. The following litmus test, IRIW ("independent reads of independent writes"), illustrates it: the write to `x` can propagate to third thread before the write to `y` does, while the write to `y` propagates to the fourth thread before the write to `x` does[2]:

IRIW

x = y = 0			
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(y,1)}$	$\texttt{r1 = load}_{\texttt{acq}}\texttt{(x);}$ $\texttt{r2 = load}_{\texttt{rlx}}\texttt{(y)}$	$\texttt{r3 = load}_{\texttt{acq}}\texttt{(y);}$ $\texttt{r4 = load}_{\texttt{rlx}}\texttt{(x)}$
$\texttt{r1 = 1} \wedge \texttt{r2 = 0} \wedge \texttt{r3 = 0} \wedge \texttt{r4 = 1}$ OK			

2.3 The C/C++11 memory model

The memory model of C++11 [18, 20] and C11 [2], which was formalised, concurrently with its development, by Batty et al. [15, 16], was the state of the art for production languages at the start of the work described in this thesis. Despite the focus of this document on its shortcomings regarding out-of-thin-air behaviour, the C/C++11 memory model achieved several of its objectives:

- It seems to almost form an envelope around hardware and compiler optimisations, and many of its shortcomings to do so seem fixable [83, 44].
- It is implementable on top of x86-TSO [15], Power [71, 13] (with a few fixes, see [44, 47]), and ARM [33], with simple and efficient compilation schemes (which we apply to our memory model in Chapter 7).
- Programs written in the fragment with only nonatomic and SC accesses and locks and free of races do not exhibit relaxed behaviour [12, §3] (this property is sometimes called "DRF-SC" or "DRF0" [3]).
- It has a usable release/acquire fragment, for which program logics have been developed [84, 82]. Moreover, the release/acquire fragment can be made optimal with respect to the standard compilation scheme with relatively minor modifications [43] (as we show for our memory model in Section 9.2).

Our remarks also apply to the memory model of OpenCL [89, 17], as it is an extension of the C/C++11 memory model, and suffers from the same issues.

[2]The first read of each thread is an acquire read to prevent it from executing out of order with the other read by the same thread, which would also allow this outcome, even without multi-copy-atomicity.

2.3.1 Description

The C/C++11 memory model is an axiomatic memory model. It is defined in phases:

- Each program is first mapped to the set of its "pre-executions". A pre-execution is a graph that represents the memory actions of a putative execution of the program, if we just consider the flow of values from reads to writes, but do not constrain what values reads can read, or where from.

  For example, a program where a thread reads from x, and then writes what it read to y is going to have (among others) a pre-execution where that thread reads 0 from x, and writes 0 to y, and a pre-execution where it reads 42 from x, and writes 42 to y, but no pre-execution where it reads 0 from x, and writes 42 to y.

  The set of pre-executions is defined by induction on the syntax, without interaction between the threads.

- Each pre-execution is then associated with the set of its "execution witnesses". An execution witness is a graph that proposes an interaction between the threads of a pre-execution that would make the pre-execution possible, in particular by providing, for each read, a write it proposes it reads from, and a coherence order.

- The set of pairs of a pre-execution and an execution witness (called a "candidate execution") is then filtered by a "consistency" predicate. The consistency predicate determines whether the execution witness justifies that the pre-execution should be considered a possible behaviour. In that case, the candidate execution is called an "execution", or a "consistent execution". This is the main part of the definition.

- In addition, the consistent executions are checked for data races and other such errors. This is done in a problematic way, as we describe in Section 9.4.1.

Because of the simple language we use (as defined in Section 2.1), we can make a few simplifications in our presentation. Memarian et al. [57] and Nienhuis et al. [62] show how to generate pre-executions for the (almost) entire C11 language.

Pre-executions

The definition of pre-executions is straightforward, but usually elided; we spell it out for clarity and completeness.

A pre-execution is a directed graph with two kinds of edges: *sb* ("sequence-before"), which represents program order, and *dd*, which represents data dependencies[3], and where nodes are labelled with memory actions (the set of which we will denote A), like $\mathsf{R}_{\mathsf{rlx}}\,\mathsf{x}\,\mathsf{v}$, a read of value v from location x, $\mathsf{W}_{\mathsf{rlx}}\,\mathsf{x}\,\mathsf{v}$, a write of value v to location x, etc. (we describe memory actions for our memory model, which are very close to that of C/C++11, in Section 4.1).

Definition 2. X is a *per-thread pre-execution* when it is of the form $\langle E, sb, data, \lambda\rangle$, where

- E is a set of *events*[4];

[3] We do not consider address dependencies; we discuss this limitation in Section 10.3.

[4] Batty et al. [15] call these "action identifiers". We call them events to make the relation to event structures more apparent.

- $sb \subseteq E \times E$;
- $dd \subseteq E \times E$;
- $\lambda : E \to A$.

Register state Pre-executions constrain the flow of values from reads to writes (but not the values or the writes reads read), and record data dependencies ('dd'). To keep track of that information, when computing pre-executions, we keep it in a register state, where each register is associated to the last value it was set to, and to the events it depends on:

Definition 3. A *C/C++11 register state* (denoted μ) is a map from registers to values and sets of events.

For simplicity, we assume the registers of each thread are initialised to zero:

Definition 4. The *initial C/C++11 register state* μ_0 is such that for all $\mathtt{r}$, $\mu_0(\mathtt{r}) = (0, \varnothing)$.

A register state induces a value for expressions, and thus tests:

Definition 5. The *value of an expression* $\mathtt{e}$ *in a register state* μ, ${}^{\mathrm{C}}[\![\mathtt{e}]\!]^{\mathrm{e}}_{\mu}$, is defined inductively by

$$\begin{aligned} {}^{\mathrm{C}}[\![\mathtt{v}]\!]^{\mathrm{e}}_{\mu} &= \mathtt{v} \\ {}^{\mathrm{C}}[\![\mathtt{r}]\!]^{\mathrm{e}}_{\mu} &= \mathrm{fst}(\mu(\mathtt{r})) \\ {}^{\mathrm{C}}[\![\mathtt{e1} + \mathtt{e2}]\!]^{\mathrm{e}}_{\mu} &= {}^{\mathrm{C}}[\![\mathtt{e1}]\!]^{\mathrm{e}}_{\mu} + {}^{\mathrm{C}}[\![\mathtt{e2}]\!]^{\mathrm{e}}_{\mu} \\ &\dots \end{aligned}$$

Definition 6. The *interpretation of a comparison operator* $\mathtt{cmp}$, $[\![\mathtt{cmp}]\!]$, is the associated mathematical operator.

Definition 7. The *interpretation of a comparison* is defined by:

$${}^{\mathrm{C}}[\![\mathtt{e}_1\ \mathtt{cmp}\ \mathtt{e}_2]\!]^{\mathrm{e}}_{\mu} = {}^{\mathrm{C}}[\![\mathtt{e}_1]\!]^{\mathrm{e}}_{\mu}\ [\![\mathtt{cmp}]\!]\ {}^{\mathrm{C}}[\![\mathtt{e}_2]\!]^{\mathrm{e}}_{\mu}$$

Register use induces data dependencies, which we then have to track:

Definition 8. *Occurrence* of a register $\mathtt{r}$ in an expression $\mathtt{e}$, $\mathtt{r} \in \mathtt{e}$, is defined inductively by

$$\begin{aligned} &\mathtt{r} \in \mathtt{r} \\ &\mathtt{r} \in \mathtt{e1} + \mathtt{e2} \quad \text{if } \mathtt{r} \in \mathtt{e1} \vee \mathtt{r} \in \mathtt{e2} \\ &\quad \dots \end{aligned}$$

The dependencies of an expression is the union of the dependencies of the registers occurring in that expression:

Definition 9. The *dependencies* of an expression $\mathtt{e}$ given a register state μ are

$${}^{\mathrm{C}}[\![\mathtt{e}]\!]^{\mathrm{r}}_{\mu} = \bigcup_{\mathtt{r} \in \mathtt{e}} \mathrm{snd}\,(\mu\,(\mathtt{r}))$$

The C/C++11 semantics of a thread $\texttt{ss}$ is given as the set of per-thread pre-executions inductively defined on $\texttt{ss}$. We use induction on the size of $\texttt{ss}$, rather than structural induction:

Definition 10. The *size of a list of statements* $\texttt{ss}$, $|\texttt{ss}|$, is defined inductively by:

$$\begin{aligned}
|[]| &= 0 \\
|\texttt{r} = \texttt{load}_{\texttt{mo}}(\texttt{x}); \texttt{ss}| &= |\texttt{ss}| + 1 \\
|\texttt{store}_{\texttt{mo}}(\texttt{x},\texttt{e}); \texttt{ss}| &= |\texttt{ss}| + 1 \\
|\texttt{lock}\ \ell; \texttt{ss}| &= |\texttt{ss}| + 1 \\
|\texttt{unlock}\ \ell; \texttt{ss}| &= |\texttt{ss}| + 1 \\
|\texttt{if}\ (\texttt{e}_1\ \texttt{cmp}\ \texttt{e}_2)\ \texttt{ss}_1\ \texttt{else}\ \texttt{ss}_2; \texttt{ss}| &= |\texttt{ss}_1| + |\texttt{ss}_2| + |\texttt{ss}| + 1 \\
|\texttt{r} = \texttt{e}; \texttt{ss}| &= |\texttt{ss}| + 1
\end{aligned}$$

Definition 11. The *set of per-thread pre-executions of a list of statements* $\texttt{ss}$ given a C/C++11 register state μ, ${}^{\text{C}}[\![\texttt{ss}]\!]^{\text{t}}_{\mu}$, is defined inductively on $|\texttt{ss}|$ by:

- The set of pre-executions for the empty thread is just the empty pre-execution:

$${}^{\text{C}}[\![[]]\!]^{\text{t}}_{\mu} = \{\langle \varnothing, \varnothing, \varnothing, \varnothing \rangle\}$$

- The set of pre-executions for a read contains, for each value v, and for each pre-execution X of the continuation of the read (where μ has been updated so that $\texttt{r}$ maps to v, to reflect that the read has read v), the pre-execution with an event corresponding to reading v, sb-before X:

$${}^{\text{C}}[\![\texttt{r} = \texttt{load}_{\texttt{mo}}(\texttt{x}); \texttt{ss}]\!]^{\text{t}}_{\mu} = \bigcup_{v \in V} \left\{ \left\langle \begin{array}{l} E \cup \{n\}, \\ sb \cup (\{n\} \times E), \\ dd, \\ \lambda \cup \{n \mapsto \mathsf{R}_{\texttt{mo}}\, \texttt{x}\, v\} \end{array} \right\rangle \middle| n \notin E \wedge \langle E, sb, dd, \lambda \rangle \in {}^{\text{C}}[\![\texttt{ss}]\!]^{\text{t}}_{\mu[\texttt{r} \mapsto (v, \{n\})]} \right\}$$

- The set of pre-executions for a write contains, for each pre-execution X of the continuation of the write, a pre-execution with an event corresponding to writing the value of $\texttt{e}$ in μ, sb-before X, and dd-after its dependencies:

$${}^{\text{C}}[\![\texttt{store}_{\texttt{mo}}(\texttt{x},\texttt{e}); \texttt{ss}]\!]^{\text{t}}_{\mu} = \left\{ \left\langle \begin{array}{l} E \cup \{n\}, \\ sb \cup (\{n\} \times E), \\ dd \cup \left(\left(\bigcup_{\texttt{r} \in \texttt{e}} \mathrm{snd}(\mu(\texttt{r})) \right) \times \{n\} \right), \\ \lambda \cup \{n \mapsto \mathsf{W}_{\texttt{mo}}\, \texttt{x}\, [\![\texttt{e}]\!]_{\mu}\} \end{array} \right\rangle \middle| n \notin E \wedge \langle E, sb, dd, \lambda \rangle \in {}^{\text{C}}[\![\texttt{ss}]\!]^{\text{t}}_{\mu} \right\}$$

- The set of pre-executions for a lock contains, for each pre-execution X of the continuation of the lock, a pre-execution with an event corresponding to locking, sb-before

X, plus a pre-execution with the lock failing:

$$
{}^{\mathrm{C}}[\![\texttt{lock}\ \ell; \texttt{ss}]\!]^{\mathrm{t}}_{\mu} =
\left\{ \left\langle \begin{array}{l} E \cup \{n\}, \\ sb \cup (\{n\} \times E), \\ dd, \\ \lambda \cup \{n \mapsto \mathsf{L}^{+}\,\ell\} \end{array} \right\rangle \middle| \begin{array}{l} n \notin E \wedge \\ \\ \langle E, sb, dd, \lambda \rangle \in {}^{\mathrm{C}}[\![\texttt{ss}]\!]^{\mathrm{t}}_{\mu} \end{array} \right\}
\cup
\left\{ \left\langle \begin{array}{l} \{n\}, \\ \varnothing, \\ \varnothing, \\ \{n \mapsto \mathsf{L}^{-}\,\ell\} \end{array} \right\rangle \right\}
$$

- The set of pre-executions for an unlock contains, for each pre-execution X of the continuation of the unlock, a pre-execution with an event corresponding to unlocking, sb-before X:

$$
{}^{\mathrm{C}}[\![\texttt{unlock}\ \ell; \texttt{ss}]\!]^{\mathrm{t}}_{\mu} =
\left\{ \left\langle \begin{array}{l} E \cup \{n\}, \\ sb \cup (\{n\} \times E), \\ dd, \\ \lambda \cup \{n \mapsto \mathsf{U}\,\ell\} \end{array} \right\rangle \middle| \begin{array}{l} n \notin E \wedge \\ \\ \langle E, sb, dd, \lambda \rangle \in {}^{\mathrm{C}}[\![\texttt{ss}]\!]^{\mathrm{t}}_{\mu} \end{array} \right\}
$$

- The set of pre-executions of a conditional is the set of pre-executions of the appropriate continuation:

$$
{}^{\mathrm{C}}[\![\texttt{if}\ (\texttt{e}_1\ \texttt{cmp}\ \texttt{e}_2)\ \texttt{ss}_1\ \texttt{else}\ \texttt{ss}_2; \texttt{ss}]\!]^{\mathrm{t}}_{\mu} =
\begin{cases} \text{if } {}^{\mathrm{C}}[\![\texttt{e}_1\ \texttt{cmp}\ \texttt{e}_2]\!]^{\mathrm{e}}_{\mu}, & {}^{\mathrm{C}}[\![\texttt{ss}_1; \texttt{ss}]\!]^{\mathrm{t}}_{\mu} \\ \text{else} & {}^{\mathrm{C}}[\![\texttt{ss}_2; \texttt{ss}]\!]^{\mathrm{t}}_{\mu} \end{cases}
$$

- The set of pre-executions of a register assignment is the set the pre-executions of its continuation, where μ has been updated:

$$
{}^{\mathrm{C}}[\![\texttt{r} = \texttt{e}; \texttt{ss}]\!]^{\mathrm{t}}_{\mu} = {}^{\mathrm{C}}[\![\texttt{ss}]\!]^{\mathrm{t}}_{\mu\left[\texttt{r} \mapsto \left({}^{\mathrm{C}}[\![\texttt{e}]\!]^{\mathrm{e}}_{\mu}, {}^{\mathrm{C}}[\![\texttt{e}]\!]^{\mathrm{r}}_{\mu}\right)\right]}
$$

Undef This definition does not cover `undef`, because the memory model of C/C++11 does not cover thread-local sources of undefined behaviour, as discussed in Section 1.2.4.

RMWs For brevity, this definition does not cover RMWs, which follow the same pattern as the rest of the construction.

This definition generates many graph-isomorphic pre-executions corresponding to the "same" pre-execution, but with different event names (and other definitions for C/C++11 suffer from the same defect). This does not matter for our purposes.

Definition 12. The *per-thread pre-executions of a thread* are its per-thread pre-executions in the initial register state:

$$
{}^{\mathrm{C}}[\![\texttt{ss}]\!]^{\mathrm{t}} = {}^{\mathrm{C}}[\![\texttt{ss}]\!]^{\mathrm{t}}_{\mu_0}
$$

Whole-program pre-executions A whole-program pre-execution is like a per-thread pre-execution, but where each event is labelled with both a memory action and a thread identifier (a natural number), and where there can be synchronisation edges between threads (*asw*, "additional synchronises with"):

Definition 13. X is a *whole-program pre-execution* when it is of the form $\langle E, sb, dd, asw, \lambda \rangle$, where

- E is a set;
- $sb \subseteq E \times E$;
- $dd \subseteq E \times E$;
- $asw \subseteq E \times E$;
- $\lambda : E \to A \times \mathbb{N}$.

In the figures, the events are labelled with the event name and the event label.

To obtain the whole-program pre-executions, C/C++11 takes the product of the per-thread pre-executions:

Definition 14. *Annotating a per-thread pre-execution* (respectively a set of per-thread pre-executions) *with a thread identifier* t (and no synchronisation):

$$\uparrow_t \langle E, sb, dd, \lambda \rangle = \langle E, sb, dd, \varnothing, \{(e, (a, t)) \mid (e, a) \in \lambda\} \rangle$$
$$\uparrow_t \mathcal{X} = \{\uparrow_t X \mid X \in \mathcal{X}\}$$

Definition 15. The *occurrence of a variable* $\mathtt{x}$ *in a statement* $\mathtt{s}$ (respectively in a list of statements $\mathtt{ss}$), $\mathtt{x} \in \mathtt{s}$ (respectively $\mathtt{x} \in \mathtt{ss}$), is defined inductively by:

$\mathtt{x} \in \mathtt{store_{mo}(x,e)}$

$\mathtt{x} \in \mathtt{load_{mo}(x)}$

$\mathtt{x} \in (\mathtt{r1}, \mathtt{r2} = \mathtt{rmw_{acqrel/acq}(x,e,e')})$

$\mathtt{x} \in (\mathtt{if}\ (\mathtt{e_1}\ \mathtt{cmp}\ \mathtt{e_2})\ \mathtt{ss_1}\ \mathtt{else}\ \mathtt{ss_2})$ if $\mathtt{x} \in \mathtt{ss_1} \vee \mathtt{x} \in \mathtt{ss_2}$

$\mathtt{x} \in \mathtt{s_1}; ...; \mathtt{s_n}$ if $\mathtt{x} \in \mathtt{s_1} \vee \cdots \vee \mathtt{x} \in \mathtt{s_n}$

Definition 16. The *pre-executions of a program* $\mathtt{ss_1} \mid\mid ... \mid\mid \mathtt{ss_n}$ are

$$^{\mathrm{C}}[\![\mathtt{ss_1} \mid\mid ... \mid\mid \mathtt{ss_n}]\!]^{\mathrm{p}} = \left\{ \left\langle \begin{array}{l} E \cup \{n_x \mid x \in Xs\}, \\ sb \cup sb', \\ dd, \\ \{(n_x, n) \mid x \in Xs \wedge n \in E\}, \\ \lambda \cup \{n_x \mapsto (\mathsf{W_{na}}\,\mathsf{x}\,0, 0) \mid x \in Xs\} \end{array} \right\rangle \;\middle|\; \begin{array}{l} \langle E, sb, dd, \varnothing, \lambda \rangle \in \mathcal{X} \wedge \\ \forall x, y \in Xs.\, x \neq y \implies n_x \neq n_y \wedge \\ \forall x \in Xs.\, n_x \notin E \wedge \\ total\_order(\{n_x \mid x \in Xs\}, sb') \end{array} \right\}$$

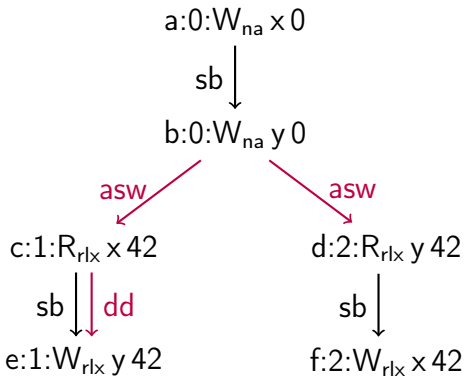
where

$$\mathcal{X} = \left\{ \begin{array}{l} \left\langle \begin{array}{l} E_1 \cup \ldots \cup E_n, \\ sb_1 \cup \ldots \cup sb_n, \\ dd_1 \cup \ldots \cup dd_n, \\ \varnothing, \\ \lambda_1 \cup \ldots \cup \lambda_n \end{array} \right\rangle \Bigg| \\ \left\langle \begin{array}{l} \langle E_1, sb_1, dd_1, \varnothing, \lambda_1 \rangle, \\ \ldots, \\ \langle E_n, sb_n, dd_n, \varnothing, \lambda_n \rangle \end{array} \right\rangle \in (\uparrow_1 {}^{\mathrm{C}}[\![\mathtt{ss_1}]\!]^{\mathrm{t}}) \times \cdots \times (\uparrow_n {}^{\mathrm{C}}[\![\mathtt{ss_n}]\!]^{\mathrm{t}}) \wedge \\ \forall i, j \in \{1, \ldots, n\}.\, i \neq j \implies E_i \cap E_j = \varnothing \end{array} \right\}$$

and

$$Xs = \{\mathtt{x} \mid \mathtt{x} \in \mathtt{ss_1} \vee \cdots \vee \mathtt{x} \in \mathtt{ss_n}\}.$$

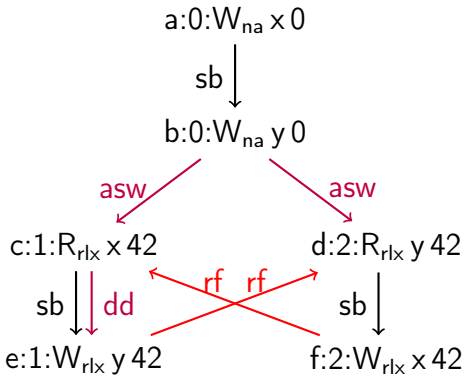
For example, LB+ctrldata+ctrl-double and LB+ctrldata+ctrl-single share the pre-execution below (among others):



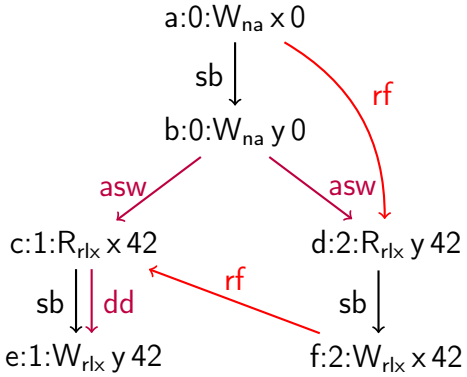
Execution witnesses

An execution witness is a proposal of a justification of "write-to-read" constraints, and of coherence. By mapping each read to a write of the same value to the same location, the *rf* relation ensures that if a program reads a value from a location, then a thread must write that value to that location at some point. The *mo* ('modification order') relation embodies coherence. The *lo* relation imposes a total, alternating order on locks and unlocks at each lock location. The *sc* relation imposes, for each location, a total order on SC accesses at that location.

For example, the diagram below superimposes an execution witness on top of the pre-execution above that justifies each each read of 42 from a location by a thread with a write of 42 to that location by the other thread. *lo* and *sc* are empty because there are no locks or SC accesses. *mo* is empty because it does not involve the initial, non-atomic writes. This example will be considered to be a consistent execution witness in the following.



The diagram below superimposes a different execution witness on top of the same pre-execution, that justifies the read of 42 from y by the initial write of 0 to y. This example will not be considered to be a consistent execution witness.



Definition 17. W is an *execution witness* when it is of the form $\langle rf, mo, lo, sc \rangle$.

Consistency

Definition 18. A *candidate execution* is the pair of a (whole-program) pre-execution and an execution witness.

See the example diagram on Page 17.

The set of candidate executions is filtered by a "consistency" predicate. We sketch the overall structure of the predicate, and some of the details, but refer to Batty et al. [15] and Batty [16] for a more thorough description.

1. The consistency predicate first ensures well-formedness of the execution witness. For example, it checks the following:

    - 'rf' is a map[5] from the reads of the pre-execution to the writes of the pre-execution. Formally, this is expressed as follows, using auxiliary definitions we

[5] A backwards map, with domain and range interchanged.

do not detail:

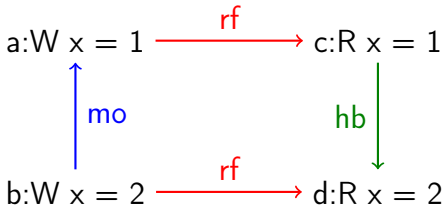
$$
\begin{array}{l}
\forall (a, b) \in rf. \\
\quad a \in E \wedge \\
\quad b \in E \wedge \\
\quad loc\_of(a) = loc\_of(b) \wedge \\
\quad is\_write(a) \wedge \\
\quad is\_read(b) \wedge \\
\quad value\_read\_by(b) = value\_written\_by(a) \wedge \\
\quad \forall a' \in E. (a', b) \in rf \implies a = a'
\end{array}
$$

- 'mo' is the union of, for each non-atomic location, a total order on the writes of the pre-execution at that location;

$$
\begin{array}{l}
(\forall (a, b) \in mo. a \in E \wedge b \in E) \wedge \\
transitive(mo) \wedge \\
irreflexive(mo) \wedge \\
\forall a \in E. \forall b \in E. \\
\quad ((a, b) \in mo \vee (b, a) \in mo) \Leftrightarrow \\
\quad \left(\begin{array}{l}
a \neq b \wedge \\
is\_write(a) \wedge \\
is\_write(b) \wedge \\
loc\_of(a) = loc\_of(b) \wedge \\
\ldots
\end{array} \right)
\end{array}
$$

- 'lo' is the union of, for each lock location, a total order on the locks and unlocks of the pre-execution at that location that alternates between lock and unlock; and
- 'sc' is a total order on SC accesses.

2. The consistency predicate then defines several derived relations. The main ones are:
   - 'sw' (synchronises-with) represents direct synchronisation, as it arises from a release write to an acquire read that reads from it, from an unlock to the next lock, etc., and including 'asw'; and
   - 'hb' (happens-before), represents inherited synchronisation; without consume reads, it is the transitive closure of the union of 'sb' and 'sw'.
3. The consistency predicate finally checks that the execution witness justifies the pre-execution. For example:
   - it checks coherence by checking for the absence of certain shapes (subgraphs) in the candidate execution, for example



- it checks that nonatomic reads read from the latest write at that location in 'hb' (we return to the consequences of this for race detection in Section 9.4.1).

It is sometimes simpler to work with immediate adjacency in mo, which we denote mo_1, than mo itself:

Definition 19.

$$(a,b) \in mo_1 \Leftrightarrow ((a,b) \in mo \wedge \nexists c.\, (c \neq a \wedge c \neq b \wedge (a,c) \in mo \wedge (c,b) \in mo))$$

For consistent executions, mo and mo_1 are interdefinable.

Definition 20. The *executions of a program* `p` are

$$[\![\mathtt{p}]\!] = \{(X,W) \mid X \in {}^{\mathrm{C}}[\![\mathtt{p}]\!]^{\mathrm{p}}, \mathit{consistent}(X,W)\}$$

Data races

Finally, C/C++11 checks for the presence of data races, mainly by looking, in consistent executions, for non-atomic accesses that are not totally ordered by hb.

2.3.2 Limitations of C/C++11

Out-of-thin-air As described in Section 1.2.2, Batty et al. [12] show that there are two programs, LB+ctrldata+ctrl-double and LB+ctrldata+ctrl-single, that share a pre-execution (see Page 17) for which the corresponding behaviour should be allowed for the first, and forbidden for the second. Because these two program share a pre-execution, the C/C++11 memory model cannot be modified to differentiate them just by changing the consistency predicate. Moreover, Batty et al. show that enriching pre-executions with "control dependencies", that is, edges from reads to memory actions inside a control block (like an `if`) the condition of which syntactically depends on the read (as in Section 1.2.2), is not sufficient either, as the two programs above have the same control dependencies.

Thread-local undefined behaviour As discussed in Section 1.2.4, the C/C++11 language definition fails to integrate the treatment of undefined behaviour and the memory model. This is illustrated by the litmus test in Section 6.7.

2.4 The Java Memory Model

2.4.1 Description of the JMM

The Java Memory Model of Manson et al. [49] (abbreviated JMM) was the second relaxed memory model for a production programming language[6]. In the JMM, an execution is justified by a sequence of partial executions of the program, where the writes of a partial execution are used to justify the reads of the next partial execution, but where the two partial executions do not have to agree on exactly which control-flow path of the program was taken. This allows the JMM to relax a partial execution by finding another partial

[6]The first was the original JMM [35, §17], about which one of its authors, Guy Steele, states that while it "broke new ground [...] in the art of programming language specification", it was "broken" and had "certain obvious technical and practical flaws, and certain other very subtle flaws" [79].

execution that, by relying on reading the same values, can write additional values, thus making taking different branches of the program possible.

For example, Java (roughly) justifies the outcome of LB+ctrldata+po (Page 16), where both reads read 42, as follows (see Figure 2.2): in the first partial execution (Figure 2.2a), both reads have to read from the initial writes. However, this allows Thread 2 (on the bottom right) to write 42 to `x`. This write is then visible in the second partial execution (Figure 2.2b) by the read of `x`, which can read 42, which allows Thread 1 (bottom left) to write 42 to `y`. This write is then visible in the third partial execution (Figure 2.2c) by the read of `y`. The actions highlighted in blue are the "committed" actions that are available to justify the next execution of the sequence.

The aim of this approach was to incrementally determine causality by exploring under what conditions an action can safely be executed early. Their solution was to consider this was when data races could be somehow resolved: "early execution of an action does not result in an undesirable causal cycle if its occurrence is not dependent on a read returning a value from a data race" [49, §4.2].

2.4.2 Limitations of the JMM

One of the design goals of the JMM was to avoid out-of-thin-air behaviour. Based on a series of litmus tests, in particular the Java Causality Tests of Pugh [68] (which we describe in Section 6.3), it seemed to achieve this goal. However, Cenciarelli et al. [26] and Ševčík and Aspinall [75] showed that some common compiler optimisations, like common subexpression elimination, are not sound in the JMM. For example, the JMM forbids the outcome of JMM CSE trap of Ševčík and Petri (on Page 46) where all reads read 42, but allows it for LB+ctrldata+ctrl-double (on Page 16), despite the fact that common subexpression elimination turns the former to the latter.

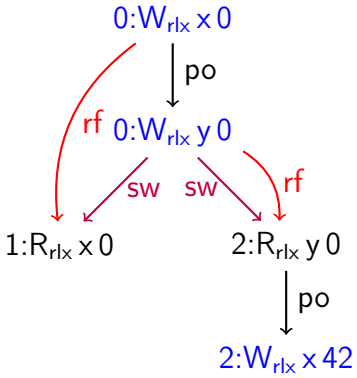
Java compilers perform these optimisations nonetheless, as shown for example by Ševčík and Aspinall for the reference "HotSpot" Java compiler [75], because compiler writers deem these optimisations more important than respecting the JMM. As a consequence, the JMM is not used as the memory model of Java. Instead, programmers have to rely on their knowledge of the compilers and hardware, as illustrated by the `concurrency-interest` mailing list [1].

Trying again to draw more general intuitive conclusions, the fact that the JMM only requires the *existence* of a sequence of partial executions is surprising, given that hardware and compilers do not consider how an individual execution arises, but rather — by looking at the program as a syntactic object — how all executions arise. There have been attempts to revisit the JMM approach by changing the justification mechanism, for example the work of Jeffrey and Riely, which we discuss in Section 11.1.

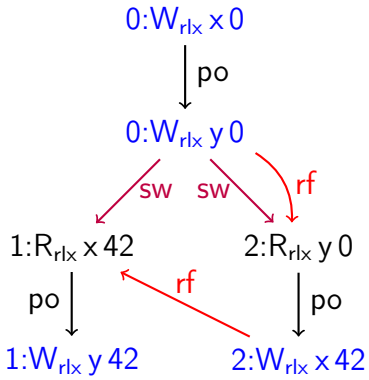
2.5 Hardware memory models

Memory models for hardware have different design constraints from programming language memory models. As the hardware executes the machine code as it finds it[7], hardware memory models hinge on *syntactic* program constructs.

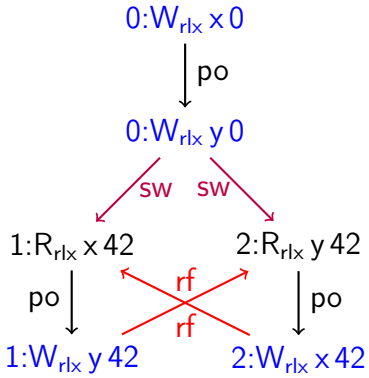
[7] or works hard to maintain the illusion that it does for sequential contexts.



(a) First partial execution: both reads read the initial value, 0.



(b) Second partial execution: the read of the second thread reads 42



(c) Final execution: both reads read 42

Figure 2.2: Sketch of the justification of the outcome of LB+ctrldata+po where both reads read 42 in the Java memory model. Memory actions that are committed after the partial execution are highlighted in blue. 'po' is program order, 'rf' is reads-from, and 'sw' is thread synchronisation.

Dependencies In the program below, because there is an *artificial dependency* between the read and the write in the first thread, hardware forbids the outcome where both reads read 42. An artificial dependency (also known as a "fake" dependency) is a syntactic dependency that does not change the values produced. In the example below, while there is an occurence of `r1` in the expression determining the value written to `y`, the expression always evaluates to `42`, independently of the value of `r1`:

LB+po-dep+data

`x = y = 0`

`r1 = load`$_{\texttt{rlx}}$`(x);`	`r2 = load`$_{\texttt{rlx}}$`(y);`
`store`$_{\texttt{rlx}}$`(y,42 + r1 - r1)`	`store`$_{\texttt{rlx}}$`(x,r2)`

`r1 = r2 = 42` OK for a programming language
BAD for hardware

However, a seemingly benign peephole optimisation that removes this seemingly useless computation turns it into the following program, for which both reads reading 42 is allowed by the Power and ARM architectures, and observed on ARMv7 hardware, because of out-of-order execution:

LB+po+data

`x = y = 0`

`r1 = load`$_{\texttt{rlx}}$`(x);`	`r2 = load`$_{\texttt{rlx}}$`(y);`
`store`$_{\texttt{rlx}}$`(y,42)`	`store`$_{\texttt{rlx}}$`(x,r2)`

`r1 = r2 = 42` OK

Branching Similarly, hardware memory models ascribe more restricted behaviour to LB+ctrldata+ctrl-double than to LB+ctrldata+po (see Page 16), operationally on the basis that the former features more branches, and architecturally because they respect control dependencies from reads to writes (as far as other threads can observe).

This situation is not completely clear-cut. Some programmers are surprised that peephole optimisations such as removing artificial data dependencies can change behaviour. Hardware architects feel a pressure to move to hardware memory models where these optimisations are sound.

One important principle of these hardware memory models is that they do not feature observable value speculation (except for branch targets, but they do not leak into values for reads and writes): values are constructed by the threads from values they have obtained from "previous" operations (immediate values in the program source, and reads from memory), with respect to a total order (execution order in operational models).

2.5.1 The Power memory model

Reference, operational model Sarkar et al. developed an operational model of Power [72, 71, 13] with the aim of faithfully capturing the architectural intent of Power, through extensive discussion with Power architects, and extensive testing against hardware [6]. This model was further refined and interfaced with the Power instruction set architecture by Gray et al. [36]. This is the model we consider in the rest of this section.

Alternative, axiomatic models Alglave et al. [7] developed an alternative, axiomatic model of Power. This model has been validated by testing against Power hardware. However, it is not known whether it is equivalent to the reference, operational model. Mador-Haim et al. [46] developed another alternative, axiomatic model of Power. However, while it has been validated by testing against Power hardware, it differs from the reference, operational model, for example on MP+lwsync+addr-bigdetour-addr (`http://diy.inria.fr/cats/cav-model/A-neg-MP+lwsync+addr-bigdetour-addr.html`), which is allowed by the operational model, but forbidden by the model of Mador-Haim et al.

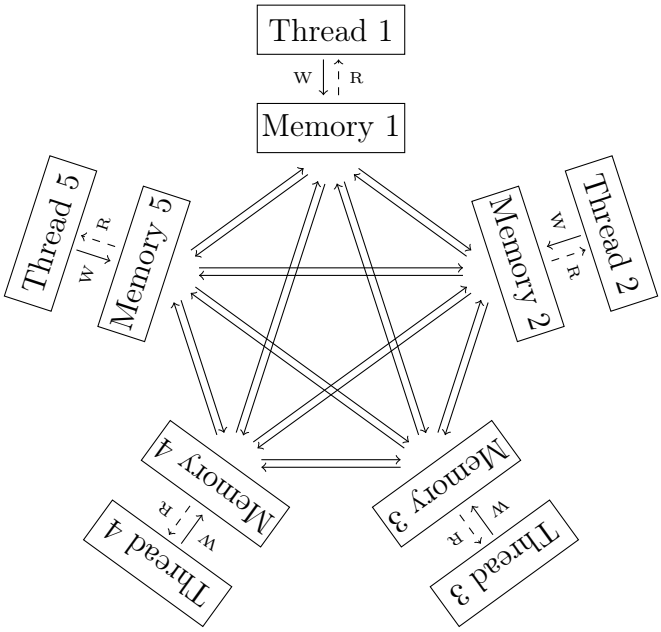


Figure 2.3: A schematic representation of the Power storage subsystem, from Maranget et al. [50].

Subsystems The operational Power memory model is split into a storage subsystem, and one thread subsystem per thread. The thread subsystems deal with thread-local effects such as out-of-order issuing of instructions, and the storage subsystem deals with IRIW-like propagation effects (illustrated in Section 2.2) between threads. Figure 2.4 lists the steps of the memory model, which we briefly describe below.

Definition 21 (Power assembly)**.** The *Power instructions* are (for our purposes) reads (`ld`) and writes (`st`) from/to memory, register operations, tests (`cmp`), conditionals (`bc`), `lwsync` ("lightweight synchronisation") and `sync` (sometimes called "heavyweight sync") memory barriers, the `isync` thread-local barrier, and load-reserve (`lwarx`) and store-conditional (`stwcx`).

Gray et al. [36] show how actual Power assembly instructions integrate with the Power memory model.

Thread subsystem

Each thread subsystem maintains a tree of *instruction instances* that can be *in-flight* or *committed*, as illustrated in Figure 2.5. An instruction instance can be added to the tree if the tree already contains an instance of the predecessor instruction. An in-flight

thread	**message**	**storage subsystem**
commit in-flight write	write request	accept write request
satisfy in-flight read	read request/response	send a read response
commit in-flight barrier	barrier request	accept barrier request
accept sync barrier ack	barrier ack	acknowledge sync barrier
fetch instruction (make it in-flight)		
forward write		
register operation		
commit read		
		propagate write
		propagate barrier
		coherence commitment

Figure 2.4: Thread subsystem and storage subsystem steps of Power (adapted from Figure 1 from Sarkar et al. [72])

branching instruction instance can have multiple instruction instance successors in the tree (for example i_5 and i_9 in Figure 2.5). An instruction instance can then be committed, provided some preconditions are met. In particular, instruction instances on which there is a dependency, for example because they determine the address or the value written, have to be committed. However, commitment does not have to be in program order, as illustrated in Figure 2.5. Instructions that follow an uncommitted branch cannot be committed. When a branch is committed, all alternative paths are removed from the tree. Moreover, before being committed, a read has to be *satisfied*, either by a write provided by the storage subsystem, or by an in-flight write *forwarded* by the thread subsystem. Control, data, and address dependencies, accessing the same location, and barriers (including `lwsync` and `sync`) constrain the execution order.

Definition 22. A *Power instruction instance* is a tuple containing a Power instruction, together with its address, and some other components.

The address of the instruction instance makes it possible to obtain the next instruction.

Definition 23. A *Power program* is a partial function from addresses (program counter) to instructions.

Definition 24. A *Power thread subsystem state* is a tuple $\langle thread, inital\_register\_state, committed\_instructions, in\_flight\_instructions, unacknowledged\_syncs \rangle$, where

- *thread* is a thread identifier;
- *inital_register_state* is a map from registers to values;
- *committed_instructions* is a set of instruction instances;
- *in_flight_instructions* is a set of instruction instances; and
- *unacknowledged_syncs* is a set of barriers.

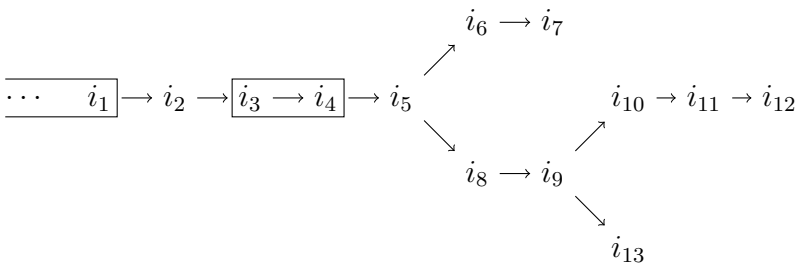


Figure 2.5: Tree of committed (in boxes) and in-flight (plain) instructions (adapted from Sarkar et al. [72]).

Storage subsystem

The storage subsystem maintains a list of the writes and barriers that have propagated to each thread. This list represents the order in which the thread has seen these events propagate. The storage subsystem also maintains a partial order on writes at each location that represents how much of coherence has been established, with a waterline of which reads have reached coherence point, on which the partial order is total, and in which no writes will be interposed. Finally, it maintains a set of syncs to be acknowledged. The storage subsystem accepts write and barrier requests from any thread that is not waiting for its sync to be acknowledged. Writes at different locations can be propagated to other threads independently, as illustrated by Figure 2.3, but respect coherence: the write has to be coherence-after all writes that were propagated to the thread to which it is propagating; however, coherence-before writes can be skipped. To determine coherence order, the storage subsystem features *coherence commitment* steps, which update the partial coherence order. The storage subsystem can also extend the coherence point waterline. All previous writes and barriers must be propagated to a thread before a barrier can be propagated to that thread. A thread that issued a sync barrier can only perform register operations until the sync has been acknowledged. To be acknowledged, the sync has to have been propagated to all threads. Reads are satisfied with the most recent write the thread has seen at that location. The `lwsync` barrier constrains propagation order, enforcing in particular that writes before it in program order are propagated before it can be propagated.

Definition 25. A *Power storage subsystem state* is a tuple $\langle$*threads*, *writes_seen*, *coherence*, *writes_past_coherence_point*, *events_propagated_to*, *unacknowledged_sync_request*$\rangle$, where

- *threads* is a finite set of thread identifiers;
- *writes_seen* is a finite set of writes;
- *coherence* is the union of, for each location, a strict partial order on the writes of *writes_seen* to that location;
- *writes_past_coherence_point* is a subset of *writes_seen* such that for each location, *coherence*, restricted to the writes of that set to that location, is a total order;
- *events_propagated_to* is a function from *threads* to lists of barriers and writes; and
- *unacknowledged_sync_requests* is finite a set of barriers.

Definition 26. A *Power system state* is a tuple $\langle thread\_states, storage\_subsystem \rangle$, where

- *thread_states* is a map from thread identifiers to Power thread states; and
- *storage_subsystem* is a Power storage subsystem state;

The program itself is threaded through in the step relation. In a more concrete model, it would be part of the storage subsystem.

2.5.2 The ARMv7 memory model

Extensionally, the ARMv7 (and the original ARMv8.0) and the Power memory models are quite close, and differ only on a few litmus tests, as discussed in Section 10.15. Moreover, ARMv7 implementations do not exploit all of the behaviour allowed by the ARM architecture, (for example, to the best of our knowledge [6], all are multiple-copy-atomic), making the two architectures even closer in practice. However, implementations differ significantly in microarchitectural flavour. Flur et al. [32] have developed a specific memory model for ARM that tries to reflect its architectural flavour, in order to gain assurance from ARM architects and communicate to programmers the desired intuitions.

The thread subsystem is very similar to the Power thread subsystem. Flur et al. [32] describe two storage subsystems, Flowing and POP.

Flowing has a more micro-architectural flavour: threads communicate by issuing memory actions that flow down through a tree of queues to a main memory that is just a map from location to values (as illustrated in Figure 2.6). Memory actions can overtake each other in the queues. Unlike in Power, reads are first-class objects in the storage subsystem: they propagate in the queues, and find their value to be satisfied either by reaching main memory, or by meeting a write at the same location in a queue.

POP ('partial order propagation') has a more abstract flavour: the storage subsystem is a partial order on the memory actions of the different threads. Flur et al. show that all the behaviours of Flowing are behaviours of POP.

2.5.3 The (revised) ARMv8 model

ARM revised the memory model of the whole ARMv8 architecture, making it multiple-copy-atomic, and confirming that artificial dependencies should be treated in the same way as non-artificial dependencies (thereby avoiding issues related to the out-of-thin-air problem for hardware). The resulting architecture, as described by Pulte et al. [69], is very close to the composition of the thread subsystem of ARMv7 with a "flat" storage subsystem (a simple map from locations to values).

2.5.4 The x86-TSO memory model

The memory model of x86, x86-TSO ("total store ordering"), as formalised by Owens et al. [63], is much stronger than that of Power and ARM, and similar to that of SPARC [39]. Threads execute in order, and the storage subsystem is rather simple (as illustrated in Figure 2.7). Each thread is equipped with an in-order, first-in first-out, buffer of writes that can individually flush, in order, to the main memory, which is just a map from location to values. Flushes can happen non-deterministically. A read reads the most recent write at that location from that thread in the thread's buffer, if there is one; otherwise, it reads from main memory. An `mfence` can only be executed when the thread's buffer is empty.

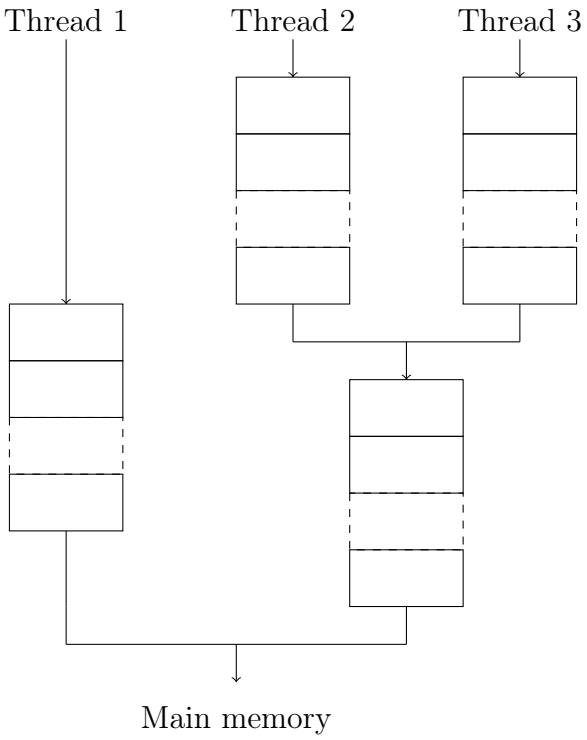


Figure 2.6: A schematic representation of one possible configuration of the Flowing ARM storage subsystem.

A compare-exchange can also only be executed when the thread's buffer is empty; if the value of the location in the main memory is the expected one, the compare-exchange changes the value to the replacement one, and succeeds; otherwise, the compare-exchange fails.

Definition 27. For our purposes, *TSO assembly* is defined inductively by

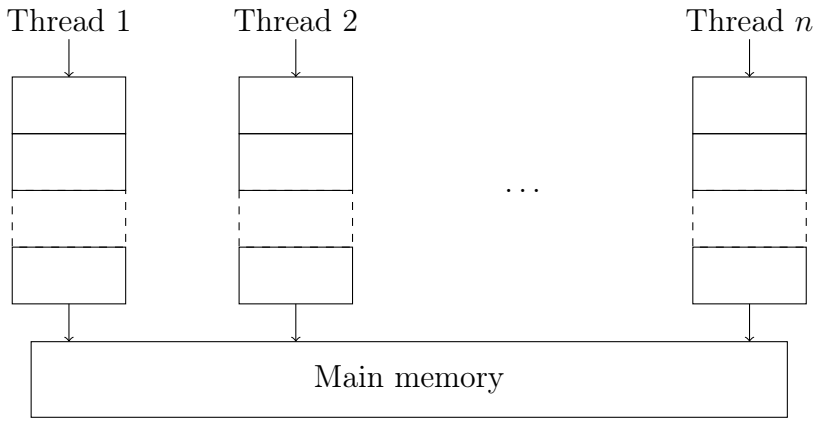


Figure 2.7: TSO main memory and thread buffers

`p`	::=	program
	`ss`$_1$ `|| ... ||` `ss`$_n$	parallel threads
`s`	::=	statement
	`r = mov x`	read
	`mov x e`	write
	`if (`e_1 `cmp` e_2`)` ss_1 `else` ss_2	conditional
	`r = e`	register assignment
	`mfence`	memory fence
	`r1, r2 = lock cmpxch x` e_1 e_2	compare-and-exchange
	`repeat ss until (`e_1 `cmp` e_2`) s`	loop
`ss`	::=	statements
	s_1`; .. ;` s_n	sequential composition

Locations `x`, registers `r`, expressions `e`, and comparisons `cmp` are as in Definition 1.

Notation We use `{ ss }` to denote `(ss)`. We also use `;` to denote concatenation.

Definition 28. b is a *TSO buffer* when it is a list of pairs of a *location* and a *value*, which are written $\mathsf{W}\,\mathtt{x}\,\mathtt{v}$, where `x` is the location, and `v` the value.

Definition 29. A *register state* (denoted μ) is a map from registers to values.

Unlike in C/C++11 (see Section 2.3), we do not track syntactic dependencies.

Definition 30. The *initial register state* μ_0 is such that for all `r`, $\mu_0(\mathtt{r}) = 0$.

A register induces a value for an expression:

Definition 31. The *value of an expression* `e` given a register state μ, $[\![\mathtt{e}]\!]_\mu$, is defined inductively by

$$[\![\mathtt{v}]\!]_\mu = \mathtt{v}$$
$$[\![\mathtt{r}]\!]_\mu = \mu(\mathtt{r})$$

$$[\![\mathtt{e}_1 + \mathtt{e}_2]\!]_\mu = [\![\mathtt{e}_1]\!]_\mu + [\![\mathtt{e}_2]\!]_\mu$$
$$\ldots$$

This can be lifted to comparisons (similarly to Definition 7), using Definition 6:

Definition 32. The *value of a comparison* is defined by:

$$[\![\mathtt{e}_1\ \mathtt{cmp}\ \mathtt{e}_2]\!]_\mu = [\![\mathtt{e}_1]\!]_\mu\ [\![\mathtt{cmp}]\!]\ [\![\mathtt{e}_2]\!]_\mu$$

Definition 33. A *TSO thread state* is the triple of a list of statements ss, a register state μ, and a buffer b.

Definition 34. A *TSO system state* is the pair of a map from a finite set of thread identifiers to TSO thread states, and of a TSO memory, that is, a map from locations to values.

Definition 35. The *initial TSO state* of a program $\mathtt{ss}_1$ `||` ... `||` $\mathtt{ss}_n$ is $\langle \{1 \mapsto \langle \mathtt{ss}_1, \mu_0, [] \rangle, \ldots, n \mapsto \langle \mathtt{ss}_n, \mu_0, [] \rangle\}, \mathit{Loc} \times \{0\} \rangle$.

Definition 36. The *TSO transition system* $\rightarrow$ is defined inductively by

$$\frac{}{\langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu, [\mathsf{W}\,\mathsf{x}\,\mathsf{v}] +\!\!+ b\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu, b\rangle\}, M[\mathsf{x} \mapsto \mathsf{v}]\rangle}$$

$$\frac{}{\langle T \uplus \{t \mapsto \langle \texttt{mfence; ss}, \mu, []\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu, []\rangle\}, M\rangle}$$

$$\frac{}{\langle T \uplus \{t \mapsto \langle \texttt{mov x e; ss}, \mu, b\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu, b +\!\!+ [\mathsf{W}\,\mathsf{x}\,[\![\mathsf{e}]\!]_\mu]\rangle\}, M\rangle}$$

$$\frac{M(\mathsf{x}) = \mathsf{v} \quad \nexists b_1, v, b_2.\, b_1 +\!\!+ [\mathsf{W}\,\mathsf{x}\,\mathsf{v}] +\!\!+ b_2 = b}{\langle T \uplus \{t \mapsto \langle \texttt{r = mov x; ss}, \mu, b\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu[\mathsf{r} \mapsto \mathsf{v}], b\rangle\}, M\rangle}$$

$$\frac{\nexists b_3, v', b_4.\, b_3 +\!\!+ [\mathsf{W}\,\mathsf{x}\,\mathsf{v'}] +\!\!+ b_4 = b_1}{\begin{array}{c}\langle T \uplus \{t \mapsto \langle \texttt{r = mov x; ss}, \mu, b_1 +\!\!+ [\mathsf{W}\,\mathsf{x}\,\mathsf{v}] +\!\!+ b_2\rangle\}, M\rangle \\ \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu[\mathsf{r} \mapsto \mathsf{v}], b_1 +\!\!+ [\mathsf{W}\,\mathsf{x}\,\mathsf{v})] +\!\!+ b_2\rangle\}, M\rangle\end{array}}$$

$$\frac{}{\begin{array}{c}\langle T \uplus \{t \mapsto \langle \texttt{r1, r2 = lock cmpxch x e1 e2}, \mu, []\rangle\}, M\rangle \\ \rightarrow \langle T \uplus \{t \mapsto (\mathtt{ss}, \mu[\mathsf{r1} \mapsto 0, \mathsf{r2} \mapsto 0], [])\}, M\rangle\end{array}}$$

$$\frac{M(x) = [\![\mathsf{e1}]\!]_\mu}{\begin{array}{c}\langle T \uplus \{t \mapsto \langle \texttt{r1, r2 = lock cmpxch x e1 e2}, \mu, []\rangle\}, M\rangle \\ \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu[\mathsf{r1} \mapsto 1, \mathsf{r2} \mapsto M(x)], []\rangle\}, M[x \mapsto [\![\mathsf{e2}]\!]_\mu]\rangle\end{array}}$$

$$\frac{[\![\texttt{e1 cmp e2}]\!]_\mu = tt}{\langle T \uplus \{t \mapsto \langle \texttt{if (e1 cmp e2) ss1 else ss2; ss}, \mu, b\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \texttt{ss1; ss}, \mu, b\rangle\}, M\rangle}$$

$$\frac{[\![\texttt{e1 cmp e2}]\!]_\mu = \mathit{ff}}{\langle T \uplus \{t \mapsto \langle \texttt{if (e1 cmp e2) ss1 else ss2; ss}, \mu, b\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \texttt{ss2; ss}, \mu, b\rangle\}, M\rangle}$$

$$\frac{}{\langle T \uplus \{t \mapsto \langle \texttt{r = e; ss}, \mu, b\rangle\}, M\rangle \rightarrow \langle T \uplus \{t \mapsto \langle \mathtt{ss}, \mu[\mathsf{r} \mapsto [\![\mathsf{e}]\!]_\mu], b\rangle\}, M\rangle}$$

$$\frac{}{\begin{array}{c}\langle T \uplus \{t \mapsto \langle \texttt{repeat ss1 until (e1 cmp e2); ss2}, \mu, b\rangle\}, M\rangle \\ \rightarrow \langle T \uplus \{t \mapsto \langle \texttt{(ss1; if (e1 cmp e2) \{ \} else repeat s until (e1 cmp e2)); ss2}, \mu, b\rangle\}, M\rangle\end{array}}$$

Chapter 3

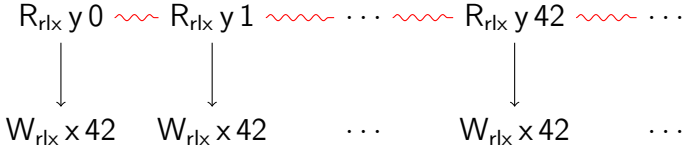
Our model, informally

3.1 Thread state

As we have seen, candidate executions, taken individually, fail to capture the difference between LB+ctrldata+ctrl-single (Page 15) and LB+ctrldata+ctrl-double (Page 16). Looking at the compiler optimisation that collapses the conditional on the second thread of LB+ctrldata+ctrl-double, we can see that it is important for the compiler to know that the write of 42 to x occurs in both control-flow paths of the conditional, or, more accurately, that it occurs irrespective of the value of the preceding read of y. This is not a per-candidate-execution property. However, the set of candidate executions almost contain the relevant information, that is, whether certain actions have a semantic dependency upon others. The missing ingredient is a way to indicate how the different candidate executions relate to each other, and the points at which they diverge into incompatible actions.

This structure, of a set of "events" equipped with some causal order and a conflict relationship, obeying certain sanity conditions, is called an event structure [61]. Event structures were introduced in 1979 as a foundational framework for "true concurrency" (that is, non-interleaving concurrency), with a single event structure describing all executions, and the conflict-free subsets of the event structure corresponding to global states of a concurrent program. We use event structures in a different way, to describe "true" concurrency, in the sense of that found in mainstream relaxed-shared-memory programming languages. Here, the current state and potential future executions of each individual thread will be represented by an event structure, the thread can take transitions that mutate that, and the whole system has a state and transitions that are the composition of those for each thread with a storage subsystem.

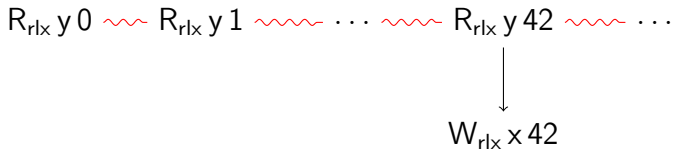
For example, the event structure of the second thread of LB+ctrldata+po (see Page 16), below, has one read event for each value that the read of y might read, and all these read events are in conflict with each other (represented by the wavy red lines), as only one of them can happen. Below each of these read events, there is the subsequent write event of 42 to y (program order is represented by black arrows):



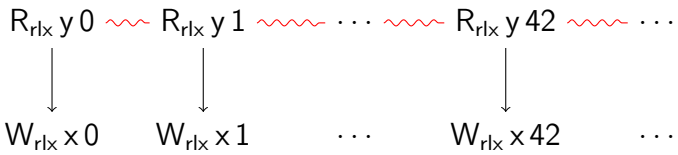
Intuitively, it is the presence of the $\mathsf{W_{rlx}\,x\,42}$ in *all* the branches of the read of y (more precisely: in the descendants of every member of the set of incompatible read events) that means that the $\mathsf{W_{rlx}\,x\,42}$ is semantically independent of the value read, and thus that it can be reordered before the read of y.

By moving to an event structure representation, we abstract over the details of how the control flow is expressed. The event structure of the second thread of LB+ctrldata+ctrl-double is the same as that of LB+ctrldata+po above, and therefore LB+ctrldata+ctrl-double has the same behaviour as LB+ctrldata+po, as desired.

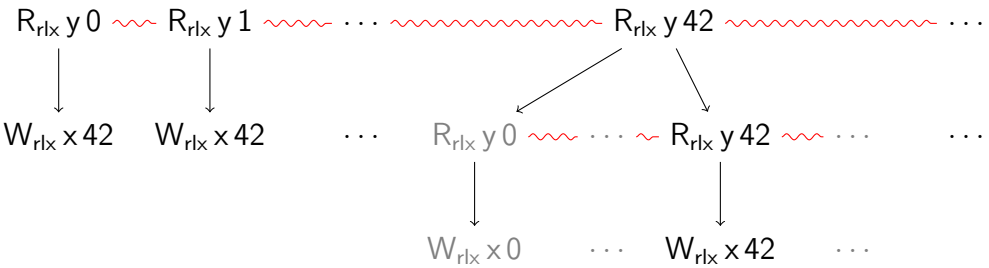
Looking at the event structure of the second thread of LB+ctrldata+ctrl-single (below), the difference with LB+ctrldata+ctrl-double, namely that the write occurs only when 42 is read, is immediate:



The difference with the second thread of LB+ctrldata+ctrl-double (below), namely that the value to write semantically depends on the value read, is also immediate.



The similarity with JMM CSE trap's litmus test below is also clear: we have to coalesce the two reads of 42, and discard the greyed out events, to make them the same:



By using event structures to represent the states of the threads, we make the relevant differences manifest in a way the dynamic semantics can easily exploit.

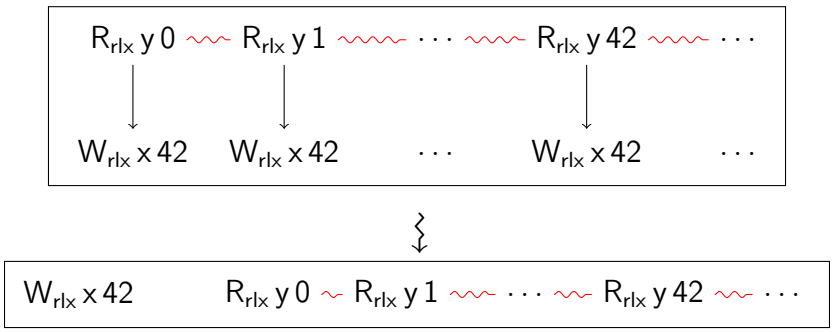
3.2 Dynamics

The goal of the memory model is to define an envelope around all "reasonable" optimisations, taking those done by current compilers to be a de facto lower bound on what should be deemed reasonable.

However, compiler optimisations as they are normally considered are highly complex syntactic transformations, typically over some intermediate representations, relying on various static analyses, and changing over time, and there are hundreds of optimisation

passes in current compilers. A semantics should not (and cannot) describe all that explicitly. Instead, we define an envelope based on steps (denoted $\rightsquigarrow$) that account for the elementary changes of memory actions that syntactic optimisation passes induce, broadly following Ševčík [86]: reordering (in fact, here, *de*ordering) and merging pairs of read and write memory actions.

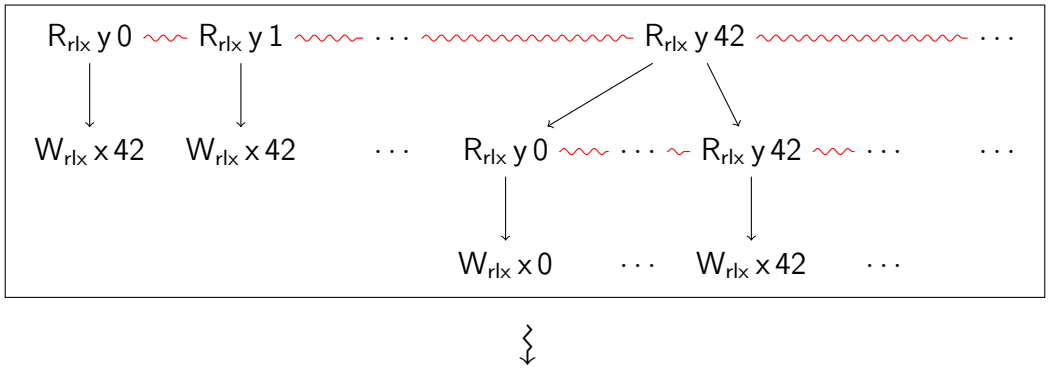
Deordering Optimisations that involve code motion rely on the fact that although the source program has a syntactic order, this order is in these cases semantically irrelevant. To account for this, we include deordering steps. For example, as there is a write of 42 in each "branch" of the read in the event structure of LB+ctrldata+ctrl-double above, there is no semantic dependency between the read and the write. The event structure makes this immediate, while in the syntax it can be obscured by control flow. Because the write does not depend on the read, and the two are at different locations, the order between the two can be removed to yield a new event structure, in a thread-local transition as below:

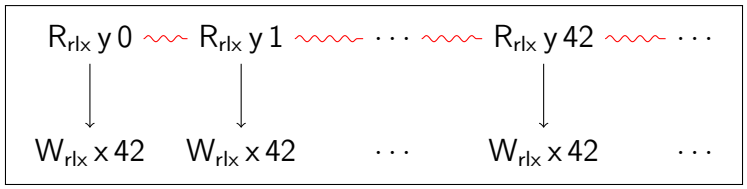


The reasoning underpinning this deordering is a semantic counterpart of what a compiler has to approximate by abstract interpretation on the syntax.

Merging Optimisations that involve removing memory instructions, like common subexpression elimination, rely on determining that the effect of one instruction can be subsumed by that of another. For example, in the JMM CSE trap litmus test, the effect of the second read of `y` is subsumed by that of the first, so the second can be eliminated and replaced by `r3 = r2`.

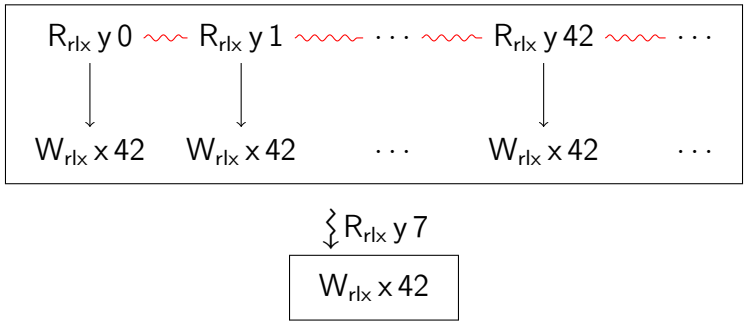
To account for this, the memory model includes merging steps (adapting Ševčík [86] from traces to event structures), in which a single event is merged into subsuming events. For example, the event structure of the second thread of the JMM CSE trap litmus test, can be turned into that of the second thread of LB+ctrldata+ctrl-double, by merging the second $\mathsf{R}_{\mathsf{rlx}}\,\mathsf{y}\,42$ into the first:





We refer to mergings by the type of memory actions involved, following Ševčík [86]: Read after Read (RaR), Write after Read (WaR), Read after Write (RaW), Write after Write (WaW), and Overwritten Write Elimination (OWE). As we detail later, the conditions of WaR and OWE are different.

Execution Steps The event structure for a thread can be mutated by optimisation steps, but it can also take execution steps, to actually perform memory actions, for any of its events that are not program-order-after any other events (the possible read values will be constrained further by the overall state, as we describe in Section 4.4.2, but for now we speak of the thread in isolation). For example, in the initial state of the second thread of LB+ctrldata+ctrl-double or LB+ctrldata+po, any of the reads can be executed, e.g. with transitions like that below:



SC accesses In sequential consistency, all threads have the same view of memory. To enforce this, the execution of an SC atomic access starts with the thread sending a request for confirmation that all other threads have received its updates. It then suspends itself until it receives an acknowledgement. Only after resuming by receiving such an acknowledgement can it execute the memory action itself.

Value-range speculation steps Optimisations are complemented by analyses which extend their applicability. For example, if analysis shows that in the program below, the read of `x` always reads a value different from 42, then the conditional can be collapsed to just $\texttt{store}_{\texttt{rlx}}\texttt{(y,42)}$.

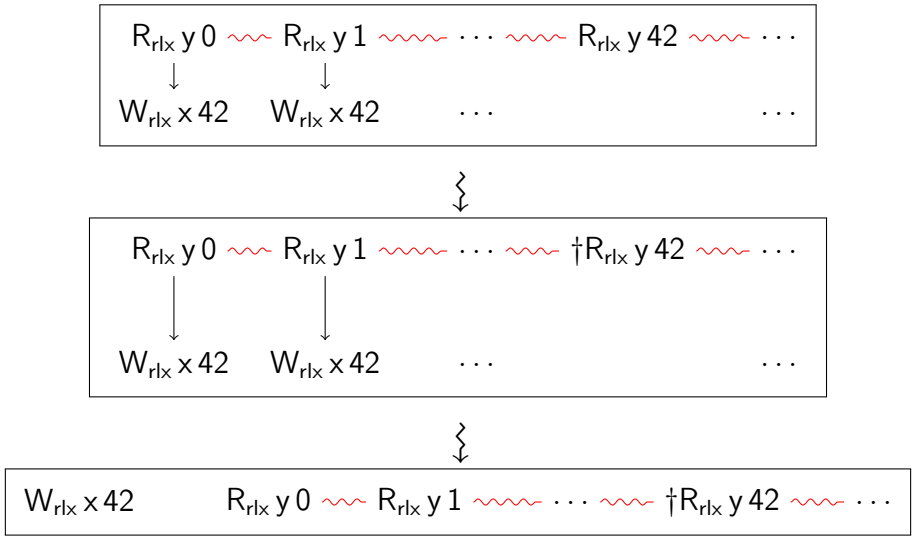
```
r1 = load_rlx(x);  ||
if (r1 != 42)      ||   ...
  store_rlx(y,42)  ||
```

This allows more behaviour, as the write to `y` can then execute out of order with the read. However, the collapse of the conditional is only valid if that extra behaviour does not invalidate the result of the analysis. For example, if the rest of the program were the second thread of LB+ctrldata+ctrl-single, this speculation would be incorrect.

To account for the effect of these analyses in the most general form we can, without fixing on any specific analyses, the memory model includes value-range speculation steps. In a value-range speculation step, a read of a certain value is speculated to be impossible,

and marked by a †. The other steps of the memory model ignore reads that have been speculated to be impossible. This speculation can be done only if the program then cannot actually execute the read of that value (this depends on the behaviour of the program as a whole, as defined below, not just the thread-local transition system that we have focussed on so far).

For the example program above, speculating that the read cannot read 42 allows deordering the write.



This mechanism allows the memory model to account for the extra optimisations enabled by inter-thread analyses, like alias analysis. This includes relational value analyses: speculation can take advantage not only of knowledge about the values for an individual location, but also about the values for several locations, and how they relate to each other. But it does come at a combinatorial price, creating a large number of potential executions even for small programs; it would be desirable to limit it if possible.

Interleaving of optimisation and execution Some optimisations only become possible when particular values become known or constrained: if a read event (recalling that this is a dynamic occurrence of a read, not a source-language syntactic occurrence) is guaranteed not to read certain values, the corresponding branches of the program do not need to be considered, and this can enable more deordering or merging steps.

This means that we cannot separate an initial optimisation phase from program execution, so instead we allow arbitrary interleaving of execution and optimisation steps. This also accommodates implementations that actually do that in practice, for example JIT compilation steps that are aware of the current value environment.

We also cannot eagerly normalise threads to "fully optimised" forms. For example, the JMM CSE trap litmus test can have two radically different executions: either the two reads are merged, which allows the write can be reordered and executed before the read, or the two reads can read different values; doing the merging optimisation eagerly would exclude the latter.

Relaxed atomics and nonatomics Relaxed atomics and nonatomic accesses are deordered in the same way. However, nonatomic accesses are treated more aggressively by mergings than relaxed atomic accesses. For example, if the accesses in the program below were atomic, it would be unsound with respect to the expected behaviour of locks

to remove the first write. Indeed, the write of 1 could be hiding a previous write, and removing it would make that previous write visible. However, as there is a nonatomic access to the same location outside of the critical section ended by `unlock l`, no race-free program can observe the write of 1 happening. Therefore, this transformation is sound for nonatomic accesses (and allowed in our memory model).

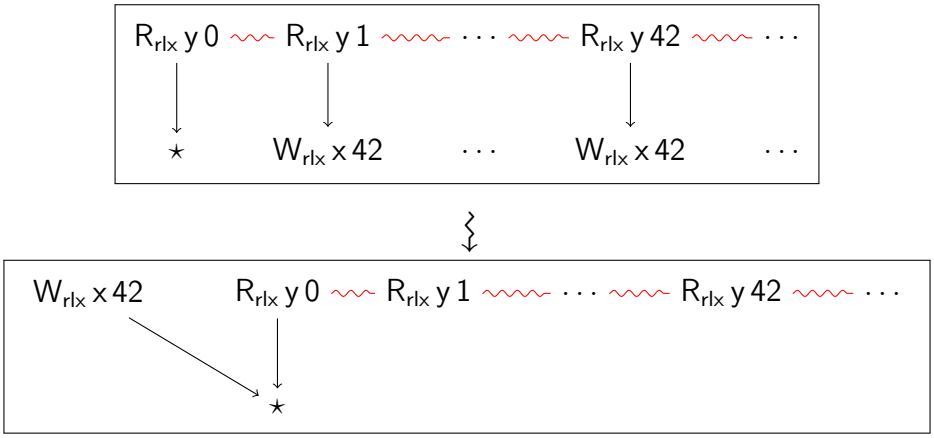
```
store_na(x,1);
unlock l;
store_na(x,2)
```

Some steps are accounted for implicitly Not all the computation and optimisation steps that runtimes and compilers perform need to be accounted for explicitly. Some do not change the event structure of a thread at all, e.g. replacing $2 + 2$ by 4. Others may radically change its memory accesses, e.g. simplifying a thread-local computation, but, if this involves only variables that (during this part of the computation) are private to the thread, it should not affect whether other optimisations across that computation are permitted.

Undefined behaviour Undefined behaviour represents partiality of the language specification. When a program point exhibits undefined behaviour, which we represent with a event labelled with $\star$ (to look like a wildcard), the semantics of the language does not have to consider what happens at that point. For example, when deordering the write in

```
r1 = load_rlx(x);
if (r1 == 0)
  undef
else
  store_rlx(y,42)
```

the branch that contains undefined behaviour does not have to be considered (in the same way that a compiler can assume that the program being compiled does not exhibit undefined behaviour):



When a $\star$-labelled event is executed, it means that the undefined behaviour can be triggered, so the whole program has undefined behaviour.

Storage subsystem Communication between threads in the model is mediated by a storage subsystem that takes care of propagation effects. The storage subsystem interacts with threads by synchronising on execution steps. The storage subsystem is based on the operational Power storage subsystem of Sarkar et al. [72] (described in Section 2.5.1), providing a non-multi-copy-atomic memory that guarantees coherence, but little more. When a thread requests a read from the storage subsystem, the read is satisfied by the writes that have propagated to the thread but are not hidden by other writes. This is the point where read values are constrained in our memory model.

While we chose to use the Power storage subsystem, our memory model could use other storage subsystems; we discuss this possibility in Section 10.15.

Chapter 4

Our model, in detail

The memory model is formalised in the executable fragment of Lem [60], a lightweight specification language based on the pure fragment of ML (see Section 6.1). In this chapter, we describe the rules in conventional non-mechanised mathematics, manually transcribed from the Lem definition[1].

4.1 Memory actions

The memory actions of our language, used to label events in the threadwise event structures and for synchronisation with the storage subsystem, are as expected from the syntax, with the addition of "dead reads", marked with a '†', which are explained in Section 4.4.10, and read-modify-writes that can either succeed or fail:

a ::=	memory action
$\mathsf{R}_{\mathsf{rmo}}\,\mathsf{x}\,\mathsf{v}$	read
$\mathsf{W}_{\mathsf{wmo}}\,\mathsf{x}\,\mathsf{v}$	write
$\mathsf{RMW}^{+}\,\mathsf{x}\,\mathsf{v}{\rightarrow}\mathsf{v'}$	read-modify-write success
$\mathsf{RMW}^{-}\,\mathsf{x}\,\mathsf{v}{\rightarrow}\mathsf{v'}\ (\mathsf{v''})$	read-modify-write failure
$\mathsf{L}^{+}\,\ell$	successful lock
$\mathsf{L}^{-}\,\ell$	unsuccessful lock
$\mathsf{U}\,\ell$	unlock
$\star$	undefined behaviour
$\dagger\mathsf{R}_{\mathsf{rmo}}\,\mathsf{x}\,\mathsf{v}$	dead read

The v and v' in $\mathsf{RMW}^{+}\,\mathsf{x}\,\mathsf{v} \rightarrow \mathsf{v'}$ and $\mathsf{RMW}^{-}\,\mathsf{x}\,\mathsf{v} \rightarrow \mathsf{v'}(\mathsf{v''})$ represent the value to replace and the value to replace with, respectively. The v'' in $\mathsf{RMW}^{-}\,\mathsf{x}\,\mathsf{v} \rightarrow \mathsf{v'}(\mathsf{v''})$ represents the value read while failing.

We conflate an elementary statement that gives rise to a memory action and the said memory action.

We adopt the terminology of Ševčík [86], and of Vafeiadis et al. [83]:

Definition 37. An action is a *release action* when it is of the form: $\texttt{store}_{\texttt{rel}}\texttt{(x,e)}$, $\texttt{store}_{\texttt{sc}}\texttt{(x,e)}$, $\texttt{r1, r2 = rmw}_{\texttt{acqrel/acq}}\texttt{(x,e1,e2)}$, or `unlock` ℓ when talking of programs,

[1] Available at `http://www.cl.cam.ac.uk/~jp622/bubbly-lem/`.

and $\mathsf{W}_{\mathsf{rel}}\,\mathsf{x}\,\mathsf{v}$, $\mathsf{W}_{\mathsf{sc}}\,\mathsf{x}\,\mathsf{v}$, $\mathsf{RMW}^{+}\,\mathsf{x}\,\mathsf{v}{\to}\mathsf{v}'$, $\mathsf{RMW}^{-}\,\mathsf{x}\,\mathsf{v}{\to}\mathsf{v}'$ (v''), or $\mathsf{U}\,\ell$ when talking of event structures.

Definition 38. An action is an *acquire action* when it is of the form: `r = load`$_{\mathsf{acq}}$`(x)`, `r = load`$_{\mathsf{sc}}$`(x)`, `r1, r2 = rmw`$_{\mathsf{acqrel/acq}}$`(x,e1,e2)`, or `lock` ℓ when talking of programs, and $\mathsf{R}_{\mathsf{acq}}\,\mathsf{x}\,\mathsf{v}$, $\mathsf{R}_{\mathsf{sc}}\,\mathsf{x}\,\mathsf{v}$, $\mathsf{RMW}^{+}\,\mathsf{x}\,\mathsf{v}{\to}\mathsf{v}'$, $\mathsf{RMW}^{-}\,\mathsf{x}\,\mathsf{v}{\to}\mathsf{v}'$ (v''), $\mathsf{L}^{+}\,\ell$, or $\mathsf{L}^{-}\,\ell$ when talking of event structures.

Definition 39. An action is *synchronising* when it is a release action or an acquire action.

Definition 40. The *strength* relation $\sqsubseteq$ on memory orders is the least reflexive and transitive relation containing `rlx` $\sqsubseteq$ `acq` $\sqsubseteq$ `sc` and `rlx` $\sqsubseteq$ `rel` $\sqsubseteq$ `sc`.

4.2 State

4.2.1 Event structures

As described informally in Section 3.1, the main component of the state of a thread is a confusion-free prime labelled event structure [61]. Because our event structures are confusion-free, we use a simplified definition based on *immediate* conflict:

Definition 41. $\mathcal{E}$ is a *confusion-free prime labelled event structure* on an alphabet (a set) A when it is of the form $\langle E, \leq, \sim, \lambda \rangle$, where

- E is the underlying set, of "events".
- λ is a labelling function from E to the alphabet A.
- $\leq$ is a partial order on E that represents program order, possibly after some transformations, such that $\forall e \in E. \{e' \in E \mid e' \leq e\}$ is finite, which expresses that threads have finite histories.
- $\sim$ is an immediate conflict binary relation between events. It is irreflexive and symmetric. At most one of a set of events in immediate conflict will be executed.
  - Immediate conflict is "almost transitive":
    $\forall e, e', e'' \in E. e \sim e' \wedge e' \sim e'' \implies e \sim e'' \vee e = e''$.
  - Immediate conflict is the first point of divergence:
    $\forall e_1, e_2 \in E. e_1 \sim e_2 \implies \{e \in E \mid e \leq e_1 \wedge e \neq e_1\} = \{e \in E \mid e \leq e_2 \wedge e \neq e_2\}$.
  - Immediate conflict respects program order: $\forall e_1, e_2, e_3 \in E. e_1 \sim e_2 \wedge e_2 \leq e_3 \implies \neg(e_1 \leq e_3)$.

In our case, the alphabet A is the set of all memory actions, as defined in Section 4.1.

Drawing convention We do not show transitively induced $\leq$ edges. We elide some of the branches of the event structures in some pictures.
It is also sometimes useful to refer to (usual, not-only-immediate) conflict:

Definition 42. *Conflict*, $\#$ is the smallest superset of $\sim$ such that

$$\forall e, e', e'' \in E. e \# e' \wedge e' \leq e'' \implies e \# e''$$

4.2.2 Construction of the initial event structure

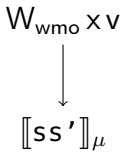
The construction of the initial event structure of a thread is a straightforward compositional function, the natural adaptation of the calculation of the set of C/C++11 pre-executions of a thread to the event structure setting.

Node names Instead of constructing a single event structure with concrete node names by renaming events structure nodes when we compose event structures, we construct an equivalence class of event structures, which differ only in their node names. Our memory model is not sensitive to node names.

Auxiliary definitions A register state μ is defined as in Definition 29, and the initial register state μ_0 as in Definition 30 (Page 43). The value of an expression and a comparison in a register state are defined in Definitions 31 and 32 (Pages 43 and 43).

Definition 43. Given a current register state μ, the equivalence class of *event structures of the memory actions of a list of statements* $\mathtt{ss}$, $[\![\mathtt{ss}]\!]_\mu$, is defined by induction on $|\mathtt{ss}|$:

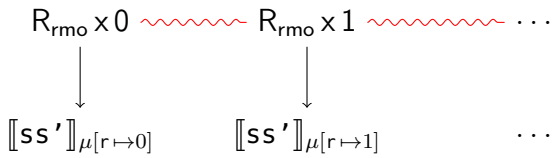
- case []: the empty event structure, $\{\langle \varnothing, \varnothing, \varnothing, \varnothing \rangle\}$.
- case $\mathtt{store}_{\mathsf{wmo}}(\mathtt{x},\mathtt{e});\ \mathtt{ss'}$: an event labelled $\mathsf{W}_{\mathsf{wmo}}\,\mathtt{x}\,[\![\mathtt{e}]\!]_\mu$, before all the events of $[\![\mathtt{ss'}]\!]_\mu$:



  formally, that is:

$$\left\{ \left\langle E \cup \{n\}, (\leq) \cup (\{n\} \times E), \sim, \lambda \cup \{n \mapsto \mathsf{W}_{\mathsf{wmo}}\,\mathtt{x}\,[\![\mathtt{e}]\!]_\mu\} \right\rangle \;\middle|\; \begin{array}{l} \langle E, \leq, \sim, \lambda \rangle \in [\![\mathtt{ss}]\!]_\mu \wedge n \notin E \end{array} \right\}$$

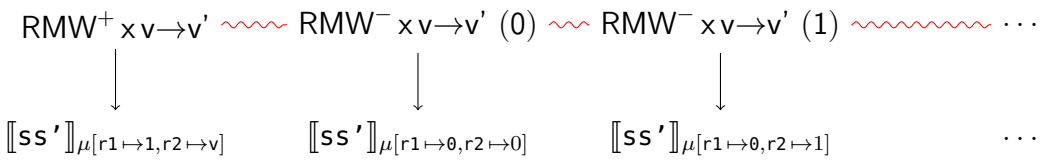
- case $\mathtt{r}\ =\ \mathtt{load}_{\mathsf{rmo}}(\mathtt{x});\ \mathtt{ss'}$: for each $\mathtt{v}$ in V, an event labelled $\mathsf{R}_{\mathsf{rmo}}\,\mathtt{x}\,\mathtt{v}$, before all the events of $[\![\mathtt{ss'}]\!]_{\mu[\mathtt{r}\mapsto\mathtt{v}]}$, with all these read events in conflict:



  formally, that is:

$$\left\{ \left\langle \begin{array}{l} \left(\bigcup_{v\in V} E_v\right) \cup \{n_v \mid v \in V\}, \\ \left(\bigcup_{v\in V} \leq_v\right) \cup \left(\bigcup_{v\in V} (\{n_v\} \times E_v)\right), \\ \left(\bigcup_{v\in V} \sim_v\right) \cup \{(n_v, n_{v'}) \mid v, v' \in V \wedge v \neq v'\}, \\ \left(\bigcup_{v\in V} \lambda_v\right) \cup \{n_v \mapsto \mathsf{R}_{\mathsf{wmo}}\,\mathtt{x}\,v \mid v \in V\} \end{array} \right\rangle \;\middle|\; \begin{array}{l} \forall v \in V.\, \langle E_v, \leq_v, \sim_v, \lambda_v \rangle \in [\![\mathtt{ss}]\!]_{\mu[\mathtt{r}\mapsto v]} \\ \forall v, v' \in V.\, n_v \notin E_{v'} \\ \forall v, v' \in V.\, v \neq v' \implies n_v \neq n_{v'} \\ \forall v, v' \in V.\, v \neq v' \implies E_v \cap E_{v'} = \varnothing \end{array} \right\}$$

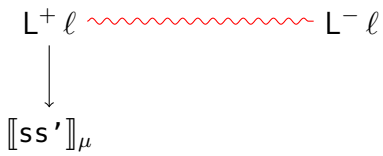
- case `r1, r2 = rmw`$_{\mathtt{acqrel/acq}}$`(x,e,e'); ss'`: for each `v"` in the domain of `x`, an event labelled $\mathsf{RMW}^{-}\,\mathtt{x}\,\mathtt{v} \to \mathtt{v'}(\mathtt{v''})$, and an event labelled $\mathsf{RMW}^{+}\,\mathtt{x}\,\mathtt{v} \to \mathtt{v'}$, with all these events in conflict, where `e` evaluates to `v` in μ, and `e'` evaluates to `v'` in μ, each before its own copy of the events of $[\![\mathtt{ss'}]\!]_\mu$:



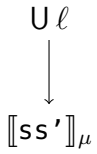
- case `(if (`$\mathtt{e}_1$ `cmp` $\mathtt{e}_2$`) `$\mathtt{ss}_1$ `else` $\mathtt{ss}_2$`); ss'`: if $[\![\mathtt{e}_1]\!]_\mu$ $[\![\mathtt{cmp}]\!]$ $[\![\mathtt{e}_2]\!]_\mu$, then $[\![\mathtt{ss}_1 +\!\!+ \mathtt{ss'}]\!]_\mu$; otherwise, $[\![\mathtt{ss}_2 +\!\!+ \mathtt{ss'}]\!]_\mu$;
- case `r = e; ss'`:

$$[\![\mathtt{ss'}]\!]_{\mu[\mathtt{r}\mapsto[\![\mathtt{e}]\!]_\mu]}$$

- case `lock` ℓ`; ss'`: two events in conflict, labelled $\mathsf{L}^{+}\,\ell$ and $\mathsf{L}^{-}\,\ell$, the former before all the events of $[\![\mathtt{ss'}]\!]_\mu$:



- case `unlock` ℓ`; ss'`: an event labelled $\mathsf{U}\,\ell$, before all the events of $[\![\mathtt{ss'}]\!]_\mu$:



- case `undef; ss'`: an event labelled $\star$:

$$\{\langle\{n\}, \varnothing, \varnothing, \{n \mapsto \star\}\rangle\}$$

.

Fences We can also define SC fences as syntactic sugar: `sc_fence` stands for an SC write of a constant to a distinguished location we ignore. See Section 9.3.

The definition above defines an equivalence class of event structures. In the rest of the semantics, we pick one representative. The behaviour does not depend on the representative.

4.2.3 Thread and program state

Definition 44. A *thread state* is a pair $\langle \mathcal{E}, s \rangle$ such that

- $\mathcal{E}$ is a confusion-free prime labelled event structure over memory actions where all events in immediate conflict are labelled with reads of different values at the same location; and
- s is a flag indicating whether the thread is suspended.

The flag is for sequentially consistent accesses, whose behaviour is defined in Section 4.4.2.

Definition 45. A *program state* is a tuple $\langle T, S \rangle$ such that

- T is a finite map from thread identifiers to thread states; and
- S is a storage subsystem state.

4.3 Induced configurations

In this section, we show the way in which an execution of our memory model induces, for each thread, a unique configuration [61] in its initial event structure, which represents its execution in the sense of what has been executed, as opposed to how, or in what order. The union of these thread-local configurations is a configuration in the disjoint union of the initial event structures of the threads, and represents the execution of the whole program.

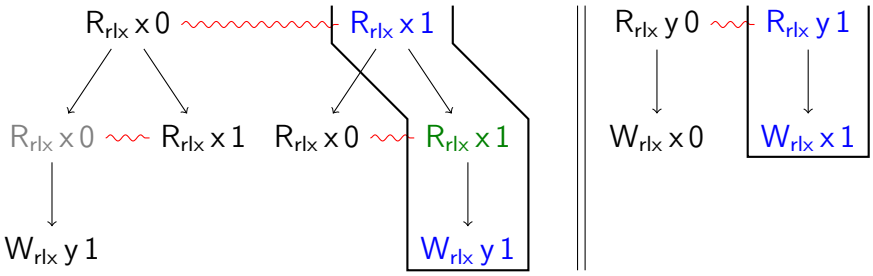
Definition 46. A *configuration* C of a confusion-free event structure $\langle E, \leq, \sim, \lambda \rangle$ is a "downwards-closed" ($\forall e \in C, e' \in E.\, e' \leq e \implies e' \in C$), conflict-free ($\forall e, e' \in C.\, e \not\sim e'$) subset of E.

Downwards in $\leq$ is upwards in our pictures.

For example, if we consider the program below

x = y = 0	
r1 = load$_{\text{rlx}}$(x);	r3 = load$_{\text{rlx}}$(y);
r2 = load$_{\text{rlx}}$(x);	store$_{\text{rlx}}$(x,r3)
if (r1 == r2)	
store$_{\text{rlx}}$(y,1)	

and an execution where the second read is merged into the first (one step per branch), and where the write is deordered, executed, and read, then the corresponding induced configuration (the inside of the boxes) is:



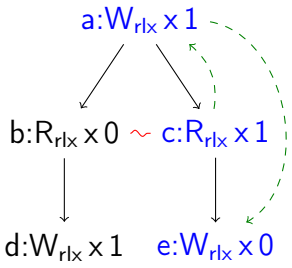
where actions that have been executed are in blue, actions that have been merged into actions that are executed, and have hence been implicitly executed, are in green, actions that have not been executed are in black, and actions that have been merged into actions that have not been not executed are in grey.

As events can be merged into other events that can themselves be merged into other events which can themselves be executed or not, what events are indirectly executed because they form part of another event is obscured. We capture this as follows:

Definition 47. The set of events that *form* an event e in an execution is the smallest set X that contains the event e itself, the events that are deordered into an event in X, and the events that are merged into an event in X.

The definitions of these steps are given in Sections 4.4.7 and 4.4.8.

In the example below, if c is merged into a, then a is merged into e, then a, c, and e form e:



Lemma 1. *Given an execution, there is a unique maximal configuration of the initial event structure of a thread (respectively program) such that all its events form executed events.*

Proof. This maximal configuration is unique because only one read of a set of reads (or RMW of a set of RMWs) in immediate conflict can be executed, or merged into another read (the others are discarded), and the reads (and RMWs) determine the maximal configuration. □

Definition 48. The *induced configuration* of the initial event structure of a thread (respectively program) associated with an execution is the unique maximal configuration such that all its events form executed events.

Reads can be executed, yet not appear in the induced configuration, when they are speculated; in that case, they are not taken into account for races (see Definition 4.4.4).

4.4 Transitions

The transitions of the program state arise from internal or combined transitions of the thread subsystems and the storage subsystem. In the description of combined steps, we distinguish the steps of the storage subsystem with a '$\diamond$' instead of a '$\bullet$'.

Definition 49. A thread step is *silent* when it does not interact with the storage subsystem (when it does not have a $\diamond$ clause).

thread t step	whole system step	storage subsystem step
normal execution step e	$t : e$	$t : e$
execution of undef	undef	
execution of dead read	cancel execution	
internal step $\tau \ldots$	$t : \tau \ldots$	
	$\tau \ldots$	internal step $\tau \ldots$
SC step	$t : \mathit{sync}$	$t : \mathit{sync}$
resumption step	$t : \mathit{sync\text{-}ack}$	$t : \mathit{sync\text{-}ack}$
VRA	$t : \dagger e$	

All thread steps require the thread to not be suspended, except the resumption step. All thread steps keep (or make, for the resumption step) the thread non-suspended, except the SC step.

4.4.1 Outcome

Definition 50. A *trace* is an alternating sequence of state and transition label, starting and ending with a state.

Definition 51. An *outcome* is either a set of traces, or undefined behaviour.

Definition 52. The *outcome of a program* P is undefined behaviour if one of its traces features a data race or executes thread-local undefined behaviour, and is the set of its traces otherwise.

4.4.2 Storage subsystem

The storage subsystem is a non-multiple-copy-atomic storage subsystem that preserves coherence, based on the Power storage subsystem of Sarkar et al. [72], as presented in Section 2.5.1. While this might seem arbitrary, this storage subsystem model is relatively simple, and is broadly the "minimal" non-multiple-copy-atomic storage subsystem that enforces coherence. Moreover, we want to expose as much of the strength of Power as possible, as C/C++11 did, and Power is the weakest target to which C/C++11 is supposed to be mapped without barriers for relaxed atomics. We discuss the consequences of this choice of storage subsystem for the implementability on ARM in Section 10.15.

Locks Our storage subsystem extends the Power storage subsystem with lock and unlock steps. They are composed of primitive Power storage subsystem steps. A lock is composed of a read immediately followed by a successful write conditional, and an unlock is composed of a write immediately followed by a lwsync. This matches the standard Power lock implementation [38] as analysed by Sarkar et al. [71, §5.1]. Because the memory model works directly on the storage subsystem state, there is no need to surround the lock with a loop: the test of whether the right value can be read can be done in the memory model, and the lock action is performed only if the right value is available. The L^{-} event corresponds to spurious failures of the lock, which we discuss in Section 10.6.

Synchronising accesses Our storage subsystem modelling of accesses that do impose some additional order follows the compilation scheme of C/C++11 to Power of McKenney and Silvera [56], which we describe in Section 7.2:

- Release writes: Release writes are modelled as an lwsync followed by the write. As the thread does not need to wait for any acknowledgement of the lwsync, these two steps can be executed atomically (their propagation is not, though).

- Acquire reads: From the point of view of the storage subsystem, acquire reads are just plain reads.

- Sequentially consistent accesses: Sequentially consistent accesses are modelled as a sync (heavyweight barrier) followed by the memory access itself. In order for the sync to take its effect, the sync needs to be acknowledged by the storage subsystem before further memory actions are issued, so the thread has to be suspended. Note that on Power, a thread that has issued a sync can still do register operations; as the event structure construction erases register operations, we can completely suspend the thread.

Read-modify-writes Read-modify-writes are modelled similarly to locks.

Races The storage subsystem keeps a trace of the executed actions to be able to detect races during execution steps, that is, the execution of two non-atomic memory accesses, at the same location, by two different threads, at least one of which is a write, and both form events of the induced configuration (otherwise, speculative reads that are simply executed early, but do not have an impact on the execution, would induce races).
This definition is kept simple by our assumption about location typing, which we discuss in Section 10.4. Without location typing, only one of the accesses needs be non-atomic, and RMWs can also be involved.

Locking issues The storage subsystem also keeps track of which thread is holding which lock, to detect locking errors: a thread locking a thread it already holds, or unlocking a thread it does not hold.

Initial writes Every location is initialised with a fully-propagated write of 0 that has reached coherence point.

4.4.3 Relativisation

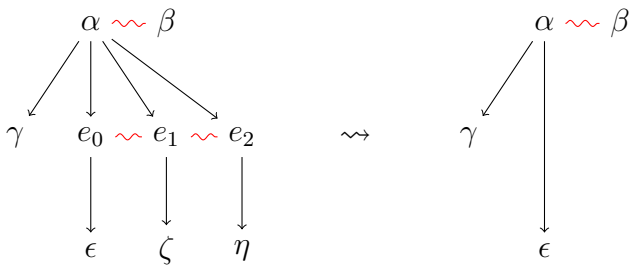
Many of the steps of the semantics are expressed using a transformation on event structures, relativisation, that, given an event of an event structure, returns the event structure where all events in conflict with that event, and the event itself, have been removed.

The exclusion of an event e, $\neg e$, is "what e excludes by itself", that is, events in immediate conflict with e, and their descendants:

Definition 53. If $\langle E, \leq, \sim, \lambda \rangle$ is a confusion-free event structure, and e an event of E, then the *exclusion* of e, $\neg e$, is $\{e' \in E \mid \exists e'' \in E, e \sim e'' \wedge e'' \leq e'\}$.

Definition 54. If $(E, \leq, \sim, \lambda)$ is a confusion-free event structure, and e is an event of E, then the *relativisation* of E with respect to e, $\mathit{relativise}(e, \langle E, \leq, \sim, \lambda \rangle)$, is $\langle E', \leq \cap E'^2, \sim \cap E'^2, \{(x, y) \in \lambda \mid x \in E'\} \rangle$, where $E' = \{e' \in E \mid e' \notin \neg e \wedge e \neq e'\}$.

For example, the relativisation of the event structure on the left with respect to e_0 is the event structure on the right. The event e_0 itself is removed, but the events below e_0 are kept; the events in immediate conflict with e_0, and the events below them, are removed.



These operations rely on the event structure being confusion-free to be well-defined.

4.4.4 Execution

A thread and the storage subsystem can synchronise to execute a memory action if the event that it is labelled with is ready to be executed, that is, if it is not program-order-after any other event. The storage subsystem constrains reads by only synchronising on values that can be read at the location of the read, and locks by only synchronising if the location is currently unlocked.

In the following step definitions, the event structure $\mathcal{E}$ of the thread taking the step is $\langle E, \leq, \sim, \lambda \rangle$.

Definition 55 (Step number 1, Execution step)**.**

A non-suspended thread can *execute* the event e when:

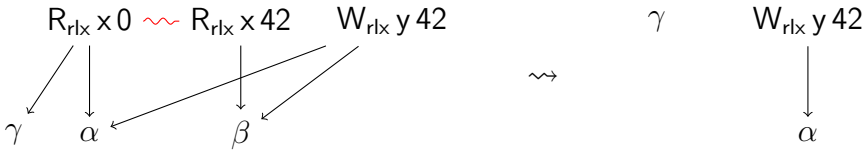
- $e \in E$;
- e is minimal with respect to $\leq$;
- e is not an SC access; and
- ◇ the storage subsystem accepts the action labelling e.

Action:

- replace $\mathcal{E}$ by $\mathit{relativise}(e, \mathcal{E})$[2];
- if the action labelling e is $\star$, then the program has undefined behaviour;
- if the last transition was also an execution transition from a different thread, where at least one of these two transitions is a write, they are both reads or writes, and they are at a nonatomic location, then there is a data race, and the program has undefined behaviour.

[2]Pichon-Pharabod and Sewell [65, §5.4] erroneously simplified this definition.

For example, if the storage subsystem accepts a request to read 0 for x in the current state (where α, β, and γ are just placeholders to make the transformation clearer), the thread below has the following execution transition:



Execution of dead reads If the action labelling e is a dead read, $\dagger\mathsf{R_{rlx}\,x\,v}$, this has special consequences (not in the execution step itself) which are described in Section 4.4.10.

4.4.5 SC accesses

The execution of SC accesses takes place in two steps: the thread synchronises, and then executes the access as a plain relaxed access.

Definition 56. The *demotion* of $\mathsf{W_{sc}\,x\,v}$ (respectively $\mathsf{R_{sc}\,x\,v}$), $\mathit{demote}(\mathsf{W_{sc}\,x\,v})$ (respectively $\mathit{demote}(\mathsf{R_{sc}\,x\,v})$) is $\mathsf{W_{rlx}\,x\,v}$ (respectively $\mathsf{R_{rlx}\,x\,v}$).

Definition 57. The *demotion of a set of events* X in an event structure $\langle E, \leq, \sim, \lambda\rangle$ is

$$\left\langle \begin{array}{l} E, \\ \leq, \\ \sim, \\ \{(x,y) \in \lambda \mid x \in E \setminus X\} \cup \{(x, \mathit{demote}(y)) \mid (x,y) \in \lambda \cap (E \times X)\} \end{array} \right\rangle$$

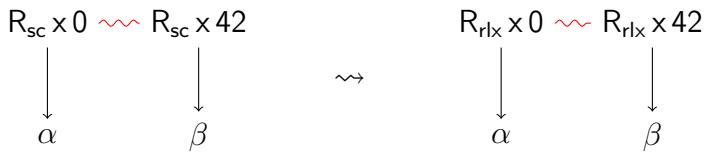
Definition 58 (Step number 2, SC step)**.**

A non-suspended thread can *issue a sync* for the set of events X when:

- X is non-empty;
- $X \subseteq E$;
- all distinct elements of X are in minimal conflict: $\forall e, e' \in X.\, e = e' \vee e \sim e'$;
- all elements of X are SC memory actions;
- the elements of X are minimal with respect to $\leq$; and
- $\diamond$ the storage subsystem accepts the *sync*.

Action: demote X, and suspend the thread.

For example, we have the following transition



if the storage subsystem accepts the request for a sync (where α and β are just placeholders to make the transformation clearer).

4.4.6 Resumption

Definition 59 (Step number 3, Resumption step)**.**

A suspended thread can *resume* when

- ◇ the storage subsystem acknowledged the sync.

Action: un-suspend the thread.

4.4.7 Deordering

We want our semantics to abstract over syntactic ordering and dependencies, and respect only semantic order and dependencies. To do so, we allow threads to remove order when this order does not matter, i.e., where the same action occurs in all the branches. The fact that the same action occurs in all the branches is represented by a set of events B below a set of events in immediate conflict above, A, or a single event above, a.

Definition 60. b is a *strict descendant* of a, $a < b$, when $a \leq b \wedge a \neq b$.

Definition 61. b is a *child* of a, $a <_1 b$, when $a < b \wedge (\neg\exists c. a \leq c \wedge c \leq b \wedge c \neq a \wedge c \neq b)$.

Definition 62. a is an *ancestor* of b when b is a descendant of a.

Definition 63 (Step number 4, Non-read deordering step)**.**

A non-suspended thread can take a *non-read deordering* step of B with respect to A when there exists a memory action a such that:

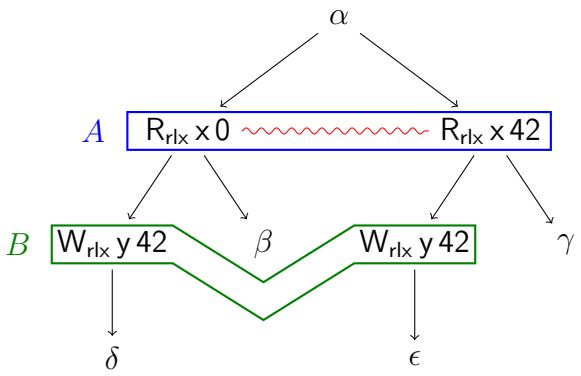
- a is not a read;
- A is a (non-empty) maximal set of events of E in immediate conflict;
- each event a_i of A either
  - has exactly one child b_i in B labelled a,
  - or, if a is not $\star$, has a child labelled $\star$, not in B,
  - or is dead;
- the memory actions of a_i and b_i are not at the same location;
- the memory action of b_i is not a release action;
- the memory action of a_i is not an acquire action; and
- all events of B are children of events of A.

Action:

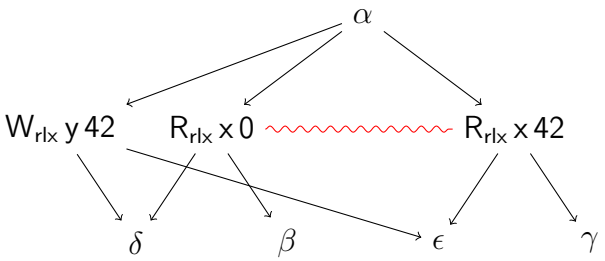
- remove all but of one the b_i, which we refer to as b below;
- remove order from A to b;
- add order from b to the descendants of the b_i;

- add order from the ancestors of the b_i, except A, to b (we discuss this side condition in Section 5.2.2); and
- restrict $\leq$, $\sim$, and λ to the updated E.

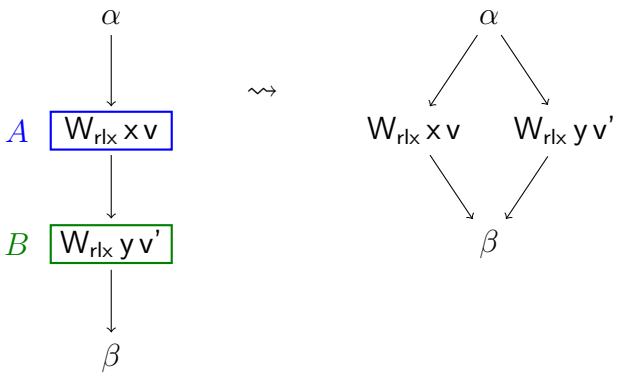
For example,



can transition to



Deordering of non-reads also applies with respect to non-reads (where A is a singleton), for example:



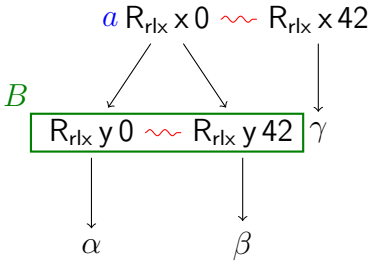
Definition 64 (Step number 5, Read deordering step)**.**

A non-suspended thread can take a *read deordering* step of a set of events B with respect to a when:

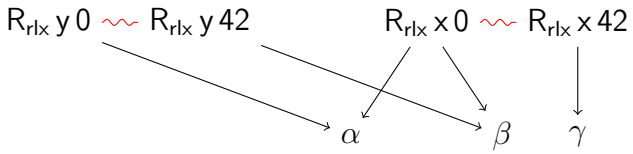
- B is a maximal set of reads of E in immediate conflict;
- a is a single event of E;
- a is not an acquire action;

- the reads of B are not SC reads; and
- a and events of B are not labelled with actions at the same location.

Action: remove order from a to B.
For example,



can transition to



Roach motel deordering The rules for deordering allow "roach motel" deordering [48], that is, extending critical sections by moving locks up and unlocks down.

4.4.8 Merging

Merging steps can merge an event into one or several others, when the effect of the latter can subsume the effect of the former. This is expressed on the event structure by doing a "partial commitment" to the subsumed event, where its alternatives are removed to express that they cannot be executed any more (as in Example 4 in §3.1).

Forwards merging An event can be merged forward (in program order) into a previous event (in the event structure) when they are at the same location and have the same value. This merging can take place at a distance, as long as there is not "too much" interposing synchronisation: for an atomic location, an interposing lock restricts the execution of the merged event; for a nonatomic location, an interposing unlock followed by a lock allows for actions from other threads to be executed in between the two events. Moreover, merging a Write into a Read is problematic [83, §7.1], so it is only allowed for nonatomic locations.

Definition 65 (Step number 6, Forward merging)**.**

A non-suspended thread can take a *forward merging* step of e_1 into e_0 when:

- e_0 and e_1 are two events of E of value v at location x;
- e_1 is not synchronising;
- $e_0 < e_1$;
- on the path between e_0 and e_1

– there are no actions at location x;

– if x has an atomic type, there is no acquire action, and it is not the case that e_0 is a write and e_1 a read;

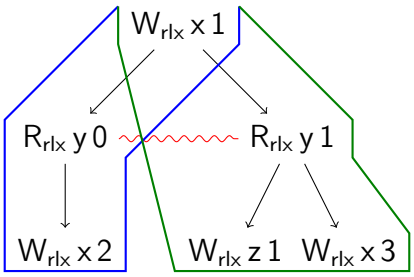
– otherwise, there is no release action followed by an acquire action, SC action, or RMW.

Action: replace $\mathcal{E}$ by $\mathit{relativise}(e_1, \mathcal{E})$.
These steps are referred to by the types of actions of e_1 and e_0: Read after Read, Read after Write, Write after Read, or Write after Write.

For example, the second read of 42 in the initial event structure of the second thread of the JMM CSE trap litmus test (in Section 3.1) can be merged into the first, transforming it into the second thread of LB+ctrldata+po.

Backward merging While the previous step is about an action being subsumed by a previous action, a write can be subsumed by a subsequent write that exists in all executions. This presence of an action in all executions is captured by checking for its presence in all configurations.

In the example below, the write of 1 to x can be merged into the writes of 2 and 3 to x, because there is an overwriting write in both configurations (in blue and green):



Definition 66. The *sub event structure from* e of an event structure $\langle E, \leq, \sim, \lambda \rangle$, $\mathit{restrict}(\langle E, \leq, \sim, \lambda \rangle, e)$, is

$$\langle E', (\leq) \cap E'^2, (\sim) \cap E'^2, \{(x, y) \in \lambda \mid x \in E'\} \rangle$$

where $E' = \{e' \in E \mid e < e'\}$.

Definition 67 (Step number 7, Backward merging)**.**

A non-suspended thread can take a *backwards merging* step of w into W when:

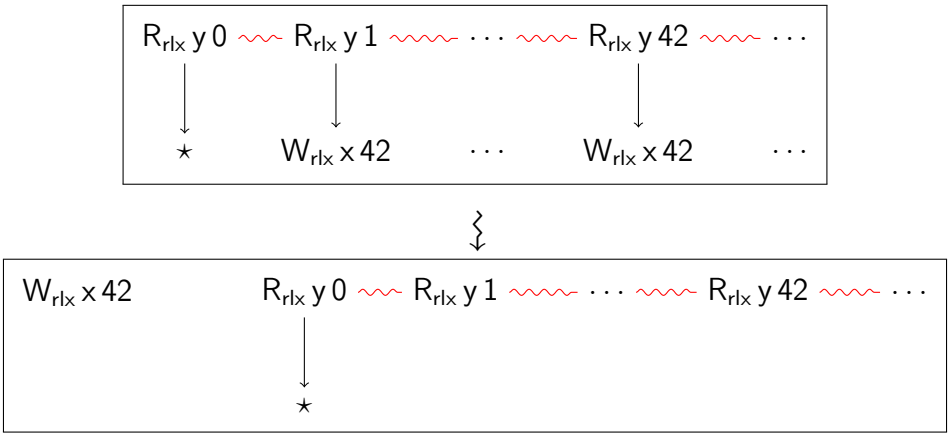
- $w \notin W$;
- w is not synchronising;
- $\{w\} \cup W$ is a set of write events of E at location x;
- each configuration of the sub-event structure rooted at w contains exactly one write in W, or a $\star$, or a dead read;
- all writes in W are in a configuration of the sub-event structure rooted at w; and

- on the path from w to any write in W:
  - there is no action at location x;
  - if x has an atomic type, there is no release action; and
  - if x has a non-atomic type, there is no release action followed by an acquire action, SC action, or RMW.

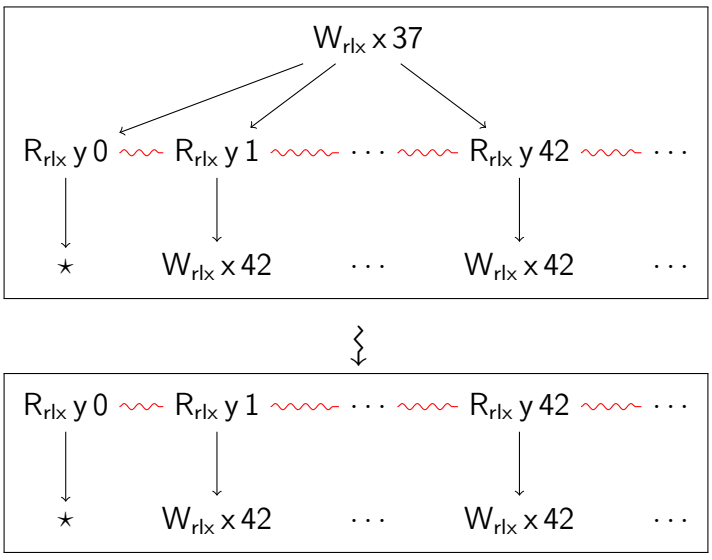
Action: replace $\mathcal{E}$ by $\mathit{relativise}(w, \mathcal{E})$.
This step corresponds to overwritten write elimination.

4.4.9 Undefined behaviour

$\star$ as a deordering joker Intuitively, if a branch triggers undefined behaviour, then behaviour is unrestricted in that branch. More formally, our memory model does not need to consider what happens in a branch that would raise undefined behaviour. A non-read deordering step does not check the branches with a $\star$-labelled child for the presence of the deordered memory action. For example, we have the following transition:



$\star$ as a backward merging joker In the same way, a $\star$ acts as a joker for backward merging. For example, we have the following transition:



$\star$ **single child** Given the above, there is no need to keep the siblings of a $\star$-labelled event, so when deordering a $\star$-labelled event, its (new) siblings and their descendants are removed.

4.4.10 Value-range speculation

Definition 68 (Step number 8, Value-range speculation step)**.**

A non-suspended thread can *speculate* that e is impossible when:

- e is an event of E;
- e is labelled with a non-dead relaxed or non-atomic read; and
- e is never executed in any execution starting from the resulting state (with the label updated as below).

Action:

- replace the label of e by the corresponding dead read label; and
- remove the sub-event structure from e from the event structure.

Dead reads as deordering jokers Dead reads are not supposed to be executed, so what takes place after them should not matter. To reflect this, the condition for the non-read deordering step (see Definition 4.4.7) with respect to a set of reads in minimal conflict does not consider dead reads in A.

Merging Similarly, for overwritten write merging, configurations that contain a dead read are ignored.

As a dead read is ignored by the steps of the memory model, there is no need to keep the sub-event-structure from the read.

Chapter 5

Discussion

In this chapter, we motivate our design choices, and make some observations about our semantics.

5.1 Design choices

Style Looking with the benefit of hindsight at the C/C++11 memory model, basing it on self-consistency of individual executions, without means of keeping track of dependencies (as Power and ARM do for *syntactic* dependencies), seems to open the door to circular/coinductive justifications, whereas we want incremental/inductive justifications, to avoid the out-of-thin-air examples that exploit exactly that failure.

To match the intuitions of how the hardware and compiler optimisations give rise to relaxed behaviour, we define our memory model as an operational semantics that constructs the execution — and therefore the values — incrementally. A read has to read a value that was *previously* (in execution order) written to memory by a write, and a write needs to appear in *all* branches of a read (that are not dead) to be deordered with respect to the read.

This is to contrast with the approach of the JMM which, while it is incremental, is incremental in a way that does not seems to relate to operational intuitions. We discuss this point further, and how the work of Jeffrey and Riely adapt the approach of the Java memory model to address this, in Section 11.1.

Storage subsystem Because we are taking an operational approach, it is easy to divide the problem in two by separating thread-local aspects from inter-thread propagation aspects. This is justified, because the issue with C/C++11 does not seem to stem from propagation problems, and the difference between hardware and programming language memory models does not seem to be changes in propagation, but changes in thread-local effects. We use the Power storage subsystem of Sarkar et al. [72] to deal with propagation effects, and can focus on the thread-local effects. When we started this work, Power was believed to have the most relaxed reasonable storage subsystem; the situation has become significantly more complicated since, as we discuss in Section 10.15.

Event structures Event structures have historically been used to make denotational models of programming languages. This is *not* the case here: we use event structures as data structures to encode some (but not all — the storage subsystem is crucial) of the internal state of an abstract machine.

Deordering We cannot simply execute the program in order, or we would miss the relaxed behaviour introduced by thread-local effects (as detailed by Nienhuis et al. [62]). C/C++11 expressed this by not imposing happens-before edges between relaxed accesses. However, this fails to identify when there are dependencies between these accesses. The JMM expressed this by allowing a read to read from a write that occurs in another branch in another execution. However, these two approaches do not reflect how hardware or compilers operate: breaking the sequential order imposed by program order is motivated by having the memory action happen in *all* branches/completions of the execution. This is what we capture with our deordering steps.

Merging Another phenomenon we have to account for is the removal of redundant memory actions by the compiler (for example common subexpression elimination) and the hardware (for example write forwarding on Power). This is the phenomenon that the JMM failed to account for (as discussed in Section 1.2.3), even though the JMM does not aim to ensure coherence. We include steps in our semantics to account precisely for this kind of optimisation. In the event structure of memory actions, we can account for fine merging of individual memory actions. For example, a single Read after Read merging step corresponds to the following transformation on the source program:

```
r1 = load_rlx(x);                 r1 = load_rlx(x);
...                               if (r1 == v) {
r2 = load_rlx(x);          ⇝        ...
...                                 r2 = r1;
                                    ...
                                  } else {
                                    ...
                                    r2 = load_rlx(x);
                                    ...
                                  }
```

This approach straightforwardly accounts for many compiler optimisations, as we show for example in Sections 8.4 and 8.5. Nonetheless, this aspect of the semantics is very ad hoc, as it bakes in the side conditions of the mergings, which are rather arbitrary set of rules, and has limitations, as discussed in Section 8.1.

Interestingly, the axiomatic approach of C/C++11 seems to be enough to abstract over those without taking special measures.

Value-range analysis Compiler optimisations are supplemented by various analyses that extend their applicability by determining certain dynamic properties of the program being compiled, for example that certain parts of the program are not executed, that certain pointers do not alias, that certain variables are constants, etc. Compiler optimisations can use this information to discard some parts of the program, to remove conditional branches, to substitute a constant for a read, etc. In our setting, this can be expressed by marking a read of a particular value as dead, and thereafter ignoring it when considering the transformations of the program. To determine whether this step can be taken, its side condition directly appeals to the whole-program semantics itself to determine whether the read is actually dead. This makes it as precise as any possible analysis, and does not put an arbitrary bound on analyses.

Where to stop? While we justify including all the steps of the semantics above, it is not clear whether they are enough to account for all the behaviours that are (or should be) introduced by compilers. We return to this question in Section 8.1.

5.2 Technical remarks

5.2.1 Deordering side condition

The side conditions of deordering enforce the following invariants, which ensure that deordering respects the aim of release/acquire pairs and SC accesses (as illustrated by the examples of Section 6.6): a memory action program-order-before a release action remains before it in the event structure, and a memory action program-order-after an acquire action remains after it in the event structure:

Lemma 2.

- *If, in a thread event structure, $a < b$, $\lambda(b)$ is a release action, a forms a', and b forms b', then $a' \leq b'$; and*
- *if, in a thread event structure, $a < b$, $\lambda(a)$ is an acquire action, a forms a', and b forms b', then $a' \leq b'$.*

Proof. By inspecting the side conditions of the deordering rules. □

The side conditions of deordering also ensure coherence: a memory action at given location program-order-before another action at the same location remains before it in the event structure (they are therefore issued in the same order, and so are in the same coherence order):

Lemma 3. *If, in a thread event structure, $a < b$, $\lambda(a)$ and $\lambda(b)$ are memory actions at the same location, a forms a', and b forms b', then $a' \leq b'$.*

Proof. By inspecting the side conditions of the deordering rules. □

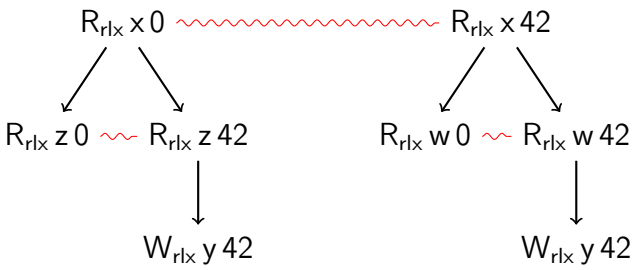
5.2.2 Deordering graph transformation

The deordering step has the following component:

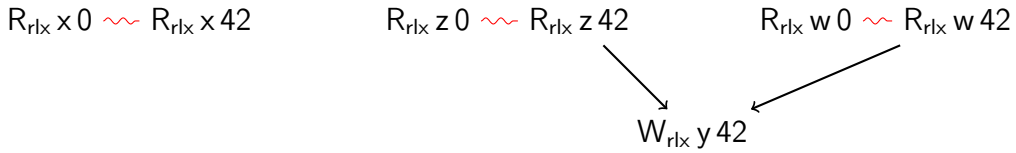
> add order from the ancestors of the b_i, except A, to b;

This was mistakenly omitted by Pichon-Pharabod and Sewell [65]. This ensures that, for example, in the following program, the write to `y` can be executed before the read of `x` only if a read of 42 for `z` and `w` can occur:

```
r1 = load_rlx(x);
if (r1 == 0)
  r2 = load_rlx(z);
  if (r2 == 42)
    store_rlx(y,42)
else
  r3 = load_rlx(w);
  if (r3 == 42)
    store_rlx(y,42)
```



The reads of z and w can be deordered with respect to the read of x, so the store to y can be deordered with respect to the read of x, but it has to remain after the reads of 42 for z *and* w.



This deordering shows that deordering does not always make the event structure more relaxed, as it requires reading 42 for both z and w.

5.2.3 SC accesses and merging

SC accesses interact with merging: before it is downgraded to relaxed, an SC access cannot be merged. Once it has been downgraded, all previous accesses have been executed, so the only merging that can occur is an overwritten write merging with a write before which there is no release action, so there is no extra behaviour: the would-be-merged write can be delayed, and only propagated together with the other write. Moreover, interposing SC accesses prevent merging, see Definitions 65 and 67.

5.2.4 No value-range speculation for synchronising reads

The only reads that can be deordered are relaxed and non-atomic reads, so speculation, which only affects execution by enabling deordering, is only useful for relaxed and non-atomic reads.

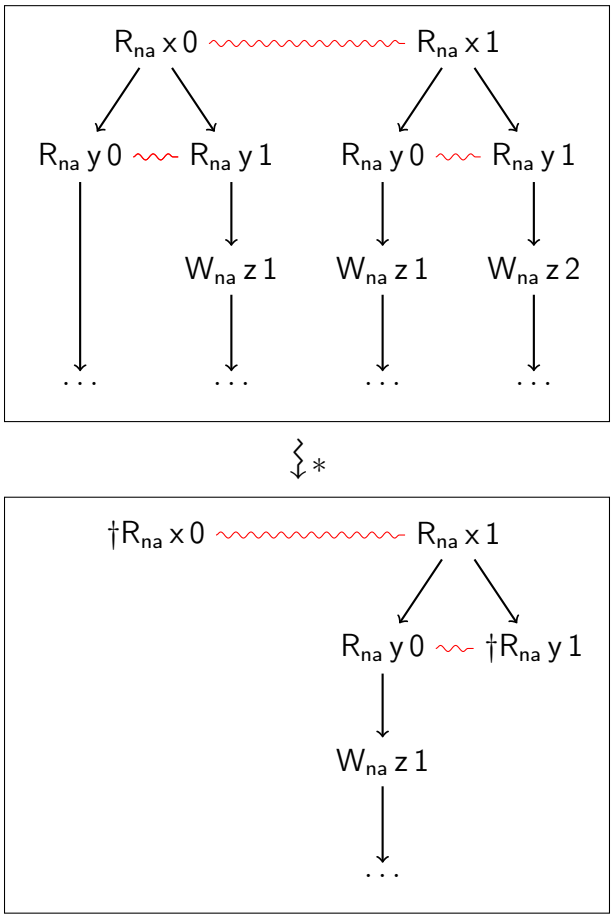
5.3 The power of value-range speculation

We introduced value-range speculation steps to express the effect of optimisations based on value-range analysis, including on-the-fly, whole-program analysis.

Value-range speculation for non-atomics For nonatomics, the effect is even more striking: at the start of a sequence of nonatomic accesses (in fact, even before), the whole nonatomic "prefix" of the event structure can be collapsed to a single line. For example, if in the following program, the nonatomic sequence starts being executed with the value of x current being 1, and that of y being 0 (as they are non-atomic locations, ignoring races, we can consider they have a current value):

```
...                          ||  ...
r1 = load_na(x);             ||  ...
r2 = load_na(y);             ||  ...
if (r1 + r2 > 0)             ||  ...
  store_na(z,r1+r2)          ||  ...
...                          ||  ...
```

then the event structure can evolve as follows, discarding all but the only possible branches for non-atomic reads:



Whether such a powerful mechanism is necessary is unclear. While no current compiler performs such an analysis in general, analyses like alias analysis or value-range propagation passes approximate this in particular cases[1], and it is not clear whether there is a middle ground.

Value-range speculation in C/C++11 The C/C++11 memory model already implicitly features an even more powerful mechanism: the per-thread component of a pre-execution is like an event structure where it has been speculated that only one path is possible, but without checking. Because there is no checking, the dangling dead reads do not need to appear in the pre-execution.

[1]See for example the interprocedural points-to analysis of GCC [87], as activated by the `-fipa-pta` flag.

5.4 Races

Our semantics only allows speculating that a read is dead if the read cannot be executed, even if there is a data race after the speculation step. This is the correct behaviour: for example, the program below is correctly identified as not being racy, even if wrongly speculating that values other than 1 cannot be read from `x` would enable a data race on `y`:

x = y = 0	
r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(x);
if (r1 == 1)	if (r2 == 1)
store$_{na}$(y,1)	store$_{na}$(y,2)

not racy

Chapter 6

Examples

The point of our semantics is to avoid out-of-thin-air behaviour. However, this cannot be proved, as there is no a priori general definition of what thin-air behaviour is. Instead, we check our semantics gives the desired behaviour on a series of examples, including those of Batty et al. [12], the Java Causality Test Cases of Pugh [68], and the JMM-breaking tests of Ševčík [74]. Because we have different aims, this can mean the desired outcomes can be different from those in that previous work.

6.1 Tool

The semantics is formalised in the executable fragment of Lem [60], and is exported to OCaml to form part of a tool which allows interactively exploring small examples, including those of this chapter. The semantics is implemented as a function that, given a state, computes the set of possible transitions from that state. The exploration tool displays the state of the semantics by displaying the event structures of the threads using Graphviz [34], and the storage subsystem state by reusing the storage subsystem state display of PPCMem [73].

The syntax of the language is suitably restricted (as discussed in Section 2.1) to make the semantics straightforwardly executable. However, the value speculation rule means that running it exhaustively is too computationally expensive except for the smallest tests. We can still explore the examples in this section with the tool by mentally excluding some steps, and restricting the value domains to the representative values (so the domains used for exploration are those represented in the figures).

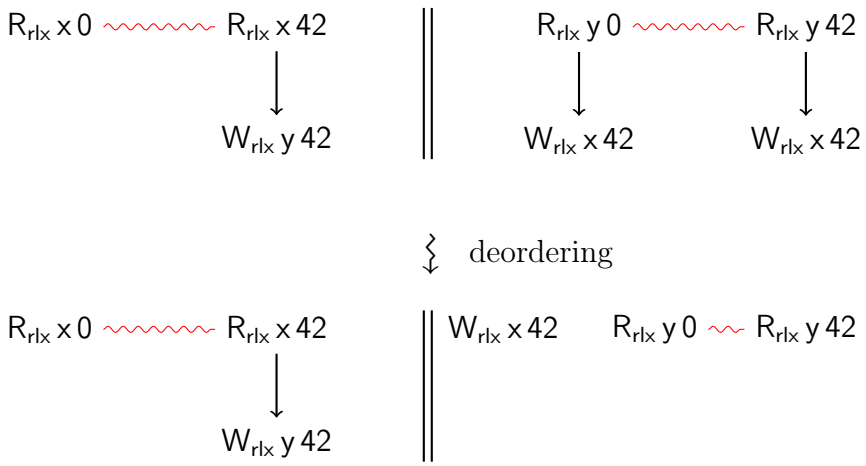
6.2 No classic out-of-thin-air

Batty et al. [12] present a series of litmus tests that explain why the C/C++11 memory model exhibits out-of-thin-air, and why it is inherent to its approach. They relate a litmus test for which a specific outcome is clearly out-of-thin-air to another for which this outcome is clearly not out-of-thin-air through a series of small modifications, but where C/C++11 cannot see the point where the change occurs (as discussed in Section 1.2.2).

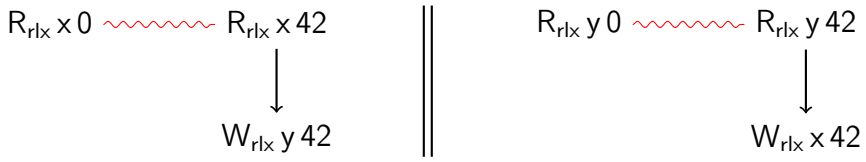
LB+ctrldata+ctrl-double (Page 16) In our semantics, from the initial state, the second thread of LB+ctrldata+ctrl-double can deorder the write of 42 to `x` with respect to the read of `y`, as the write occurs in all branches, and then execute it. The write can

then propagate to the first thread, and be read. The write of 42 to x can now be executed, propagated to the second thread, and read.

The key deordering step is:



LB+ctrldata+ctrl-single (Page 15) On the other hand, the second thread of the LB+ctrldata+ctrl-single litmus test cannot deorder the write, because it does not occur in all branches of the read in the initial event structure (it occurs in only one):



The only possible steps for both reads are to read 0, and to speculate that certain values other than 0 (which can obviously be read) cannot be read. However, as the 0 branch does not contain any write, taking speculation steps will not enable more deordering, so the reads can only read 0.

LB+ctrldata+po (Page 16) The event structure of LB+ctrldata+po is the same as that of LB+ctrldata+ctrl-double, so both reads can read 42.

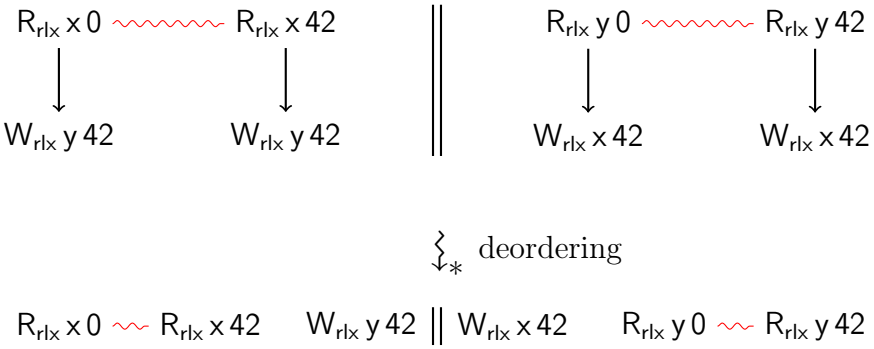
LB+ctrl+po (Page 17) The event structure of LB+ctrl+po is the same as that of LB+ctrldata+po, and thus of LB+ctrldata+ctrl-double, so both reads can read 42.

LB If there are unconditional writes of 42 by both threads, then either of the threads can deorder its write with respect to its read, and execute the write, enabling both reads to read 42.

x = y = 0

r1 = $load_{rlx}$(x);	r2 = $load_{rlx}$(y);
$store_{rlx}$(y,42)	$store_{rlx}$(x,42)

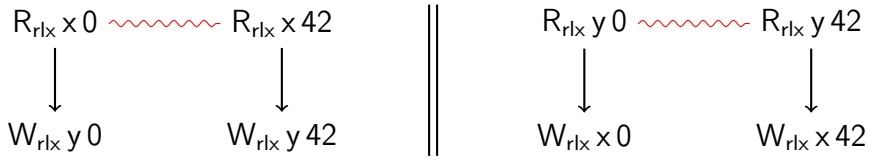
r1 = r2 = 42 OK



LB+datas If there are (actual) data dependencies from the read of each thread to its write, then the writes are of different values in some of the different branches of the read; moreover, speculation cannot mark the reads of 0 as dead, so no deordering is possible. Therefore, the first memory action to be executed has to be one of the reads, and it thus reads 0.

LB+datas

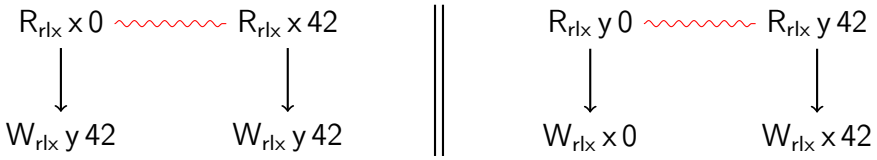
x = y = 0	
r1 = load_{rlx}(x);	r2 = load_{rlx}(y);
store_{rlx}(y,r1)	store_{rlx}(x,r2)
r1 = r2 = 42 BAD	



LB+ctrls This is similar to the previous example. If there are (actual) control dependencies from the read of each thread to its write, then there is a write only in the branch where the read read the appropriate value. Speculation cannot mark the reads of 0 as dead, so no deordering is possible. Therefore, the first memory action to be executed has to be one of the reads, and it thus reads 0.

x = y = 0	
r1 = load_{rlx}(x);	r2 = load_{rlx}(y);
if (r1 == 42)	if (r2 == 42)
store_{rlx}(y,42)	store_{rlx}(x,42)
r1 = r2 = 42 BAD	

LB+po+data (Page 37) If the write in the first thread is unconditional, but there is a data dependency from the read of the second thread to its write, then the writes are of the same value in the different branches of the read of the first thread, so the write by the first thread can be deordered, making the outcome possible.



LB+po-dep+data (Page 37) The construction of the event structure resolves the value of the expressions, and erases the dependency of the write to `y` on the read of `x`, so the event structure is the same as that of LB+po+data.

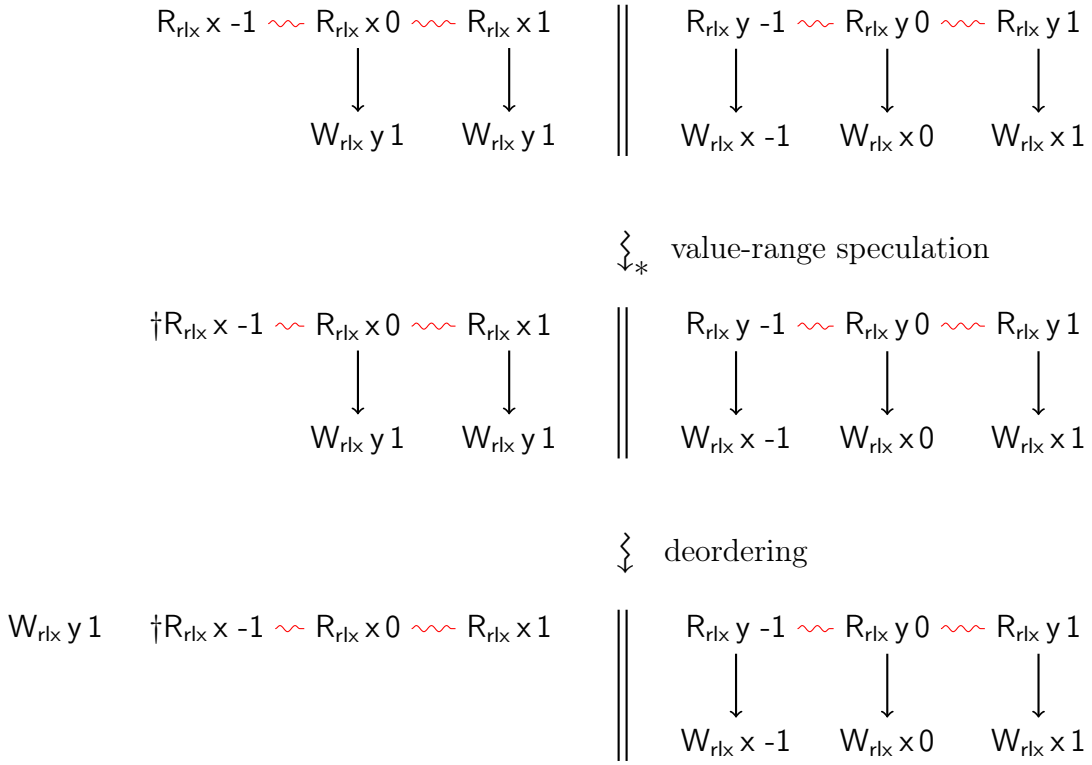
6.3 Java Causality Test Cases

Pugh presents a series of 20 "causality test cases" collected during the JSR-133 Java Memory Model development [68]. Leaving aside the two tests that involve loops and `volatile`, and with caveats for the two tests that involve `join`, the outcomes given by our memory model are the same as those desired for the JMM, except for causality test 16. This test exposes the (intentional) lack of coherence for non-volatile accesses in the Java memory model, in contrast to C/++11 relaxed atomics, which feature coherence. This series of tests exercise the power of value-range speculation in our memory model.

Causality test 1 Pugh justifies this behaviour by "interthread compiler analysis" determining that `x` and `y` cannot be negative, so `r1 >= 0` is always true.

x = y = 0	
r1 = $\texttt{load}_{\texttt{rlx}}$(x);	r2 = $\texttt{load}_{\texttt{rlx}}$(y);
if (r1 >= 0)	$\texttt{store}_{\texttt{rlx}}$(x,r2)
$\texttt{store}_{\texttt{rlx}}$(y,1)	
r1 = r2 = 1 OK	

In our semantics, value-range speculation allows marking the branches where negative values are read as dead, which enables a deordering of the write of 1 to `y`.



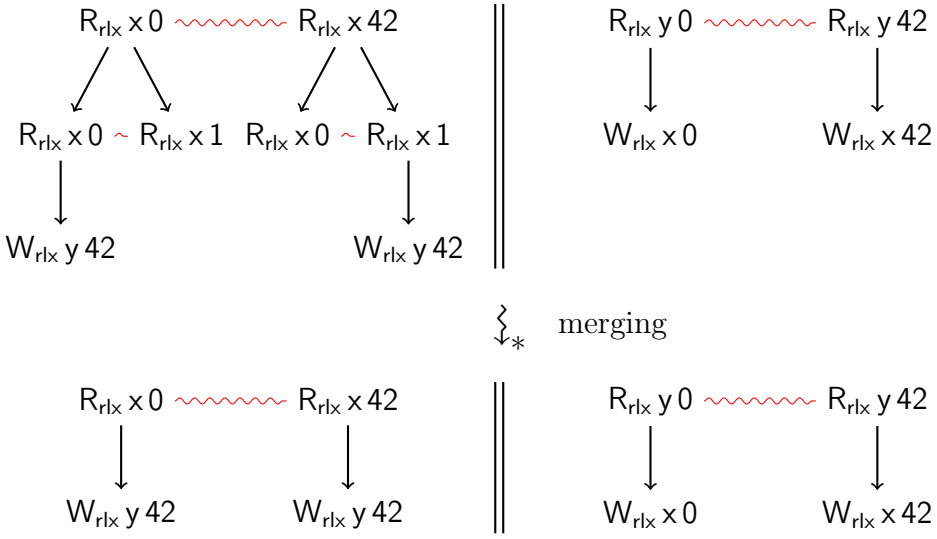
Causality tests 2 and 3 Pugh justifies this behaviour by "redundant read elimination".

x = y = 0	

r1 = load$_{rlx}$(x);
r2 = load$_{rlx}$(x);
if (r1 == r2)
  store$_{rlx}$(y,1)
	r3 = load$_{rlx}$(y);
	store$_{rlx}$(x,r3)

r1 = r2 = r3 = 1 OK	

In our semantics, Read-after-Read merging of the second read of `x` into the first enables deordering the write of 1 to `y`. Causality test 3 is causality test 2 with an additional thread executing `store`$_{rlx}$`(x,2)`.



Causality test 4 This is the same as LB+datas, which is discussed in Section 6.2.

Causality test 5 For a multiple-executions model like the JMM, there is the worry that the write of 1 to `x` by the fourth thread in an execution could somehow leak to the execution where the fourth thread writes 0, making the first two threads exhibit out-of-thin-air behaviour.

`x = y = z = 0`

`r1 = load`$_{\texttt{rlx}}$`(x);`	`r2 = load`$_{\texttt{rlx}}$`(y);`	`z = 1`	`r3 = load`$_{\texttt{rlx}}$`(z);`
`store`$_{\texttt{rlx}}$`(y,r1)`	`store`$_{\texttt{rlx}}$`(x,r2)`		`store`$_{\texttt{rlx}}$`(x,r3)`

`r1 = r2 = 1` $\wedge$ `r3 = 0` BAD

However, in our semantics, there is no danger of writes from executions leaking into other executions.

If the third and the fourth threads are replaced by a thread the code of which is the sequencing of that of the fourth thread after the third thread, then this behaviour becomes allowed. Causality tests 19 and 20, discussed below, explore this sequencing. We return to sequencing of threads in Section 10.9.

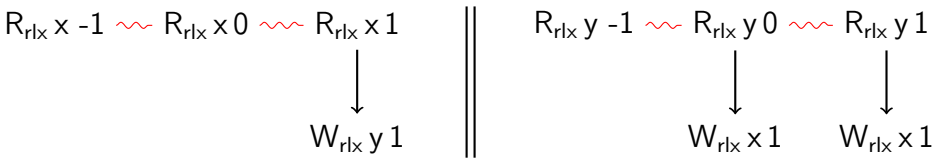
Causality test 6 Pugh justifies this by "intrathread analysis".

`x = y = 0`

`r1 = load`$_{\texttt{rlx}}$`(x);`	`r2 = load`$_{\texttt{rlx}}$`(y);`
`if (r1 == 1)`	`if (r2 == 1)`
`  store`$_{\texttt{rlx}}$`(y,1)`	`  store`$_{\texttt{rlx}}$`(x,1)`
	`if (r2 == 0)`
	`  store`$_{\texttt{rlx}}$`(x,1)`

`r1 = r2 = 1` OK

In our semantics, as in causality test 1, all reads of `y` except those of 0 and 1 can be speculated to be dead, and the write of 1 to `x` can be deordered.



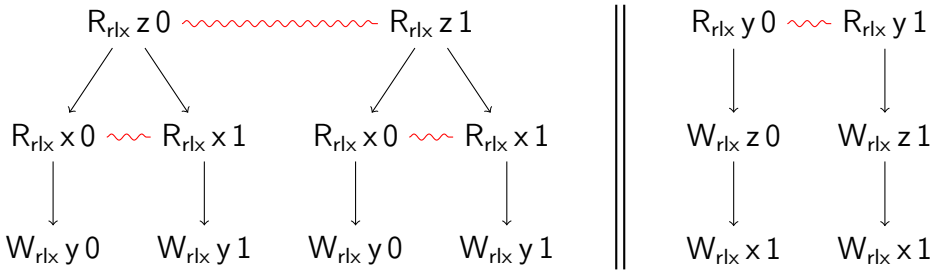
Causality test 7 Pugh justifies this as follows:

> "Intrathread transformations could move `r1 = z` to after the last statement in thread 1, and `x = 1` to before the first statement in thread 2."

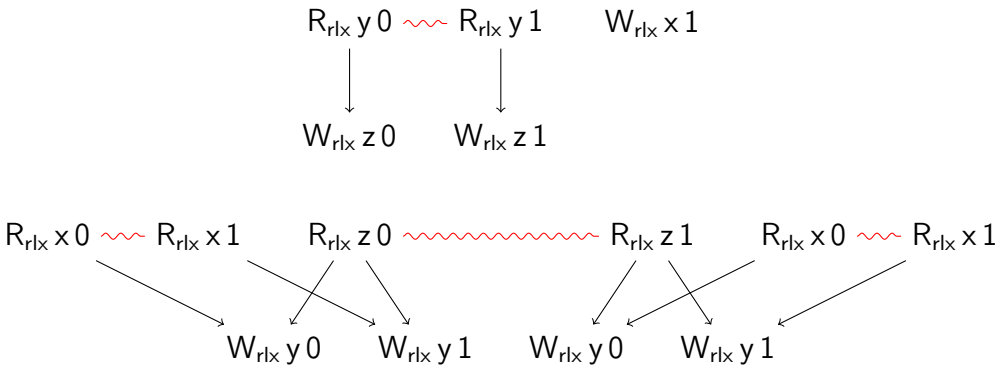
`x = y = z = 0`

`r1 = load`$_{\texttt{rlx}}$`(z);`	`r3 = load`$_{\texttt{rlx}}$`(y);`
`r2 = load`$_{\texttt{rlx}}$`(x);`	`store`$_{\texttt{rlx}}$`(z,r3);`
`store`$_{\texttt{rlx}}$`(y,r2)`	`store`$_{\texttt{rlx}}$`(x,1)`

`r1 = r2 = r3 = 1` OK



The store of 1 to x can be deordered (as below) and executed, and the reads from x can be deordered (as below) and read 1, which allows the write of 1 to y to be deordered and executed, which allows the read of y to read 1, which allows the write to z to write 1, which allows the read of z to read 1.



Causality test 8 Again, Pugh justifies allowing this test by intrathread analysis.

x = y = 0

r1 = load$_{rlx}$(x);	r3 = load$_{rlx}$(y);
r2 = 1 + r1*r1 - r1;	store$_{rlx}$(x,r3);
store$_{rlx}$(y,r2)	

r1 = r2 = 1 OK

Value-range speculation allows marking all the reads of values other than 0 and 1 as dead, which allows deordering the write to y, which is a write of 1 for both the read of 0 and the read of 1.

Causality test 9 Pugh justifies allowing this test, when the read of x by the first thread is assumed to not be able read from the write of 2 to x by the third thread.

x = y = z = 0

r1 = load$_{rlx}$(x);	r3 = load$_{rlx}$(y);	store$_{rlx}$(x,2) // after r1
r2 = 1 + r1*r1 - r1;	store$_{rlx}$(x,r3);	
store$_{rlx}$(y,r2)		

r1 = r2 = 1 OK

If we know that the read of x by the first thread cannot read any value other than 0 and 1, then the reasoning of causality test 8 applies.

Causality test 10 Pugh considers this test in the same way as test case 5, and so do we.

x = y = z = 0			

r1 = load$_{rlx}$(x);
if (r1 == 1)
  store$_{rlx}$(y,1)
	r2 = load$_{rlx}$(y);
	if (r2 == 1)
	  store$_{rlx}$(x,1)
		store$_{rlx}$(z,1)	r3 = load$_{rlx}$(z)
			if (r3 == 1)
			  store$_{rlx}$(x,1)

r1 = r2 = 1 $\wedge$ r3 = 0 BAD			

Causality test 11 Pugh considers this test in the same way as test case 7, and so do we.

x = y = z = 0	

r1 = load$_{rlx}$(z);
store$_{rlx}$(w,r1);
r2 = load$_{rlx}$(x);
store$_{rlx}$(y,r2);
store$_{rlx}$(y,1)
	r4 = load$_{rlx}$(w);
	r3 = load$_{rlx}$(y);
	store$_{rlx}$(z,r3);
	store$_{rlx}$(x,1);
	store$_{rlx}$(x,1)

r1 = r2 = r3 = r4 = 1 OK	

Causality test 12 This causality test uses arrays. To discuss it, we consider an extension of our memory model with arrays of integers in the "straightforward" way, that is array cells are treated like usual locations.

x = y = 0 $\wedge$ a[0] = 1 $\wedge$ a[1] = 2	

r1 = load$_{rlx}$(x);
store$_{rlx}$(a[r1],0);
r2 = load$_{rlx}$(a[0]);
store$_{rlx}$(y,r2)
	r3 = load$_{rlx}$(y);
	store$_{rlx}$(x,r3)

r1 = r2 = r3 = 1 BAD	

The read of `a[0]` can read 1 in the branch where the read of `x` reads 1, but this cannot happen in the branch where the read of `x` reads 0, so the write to `y` cannot be deordered, so the reads of `x` and `y` have to read 0. This execution has two reads from `a[0]`: the one in the branch where the read of `x` reads 0, and one in the branch where it reads 1. The first reads 0, while the second reads 1. However, only the first is really part of the execution, the other one is speculative (see Section 4.3), so the read of `a[0]` reads 0.

Causality test 13 This is the same as LB+ctrls, with 1 instead of 42 (see Section 6.2).

Causality tests 14 and 15 These causality tests use Java's `volatile`, which has no direct equivalent in our memory model, although it is related to sequentially consistent accesses; it also uses loops, which we do not support in our formal development, but could consider adding, as discussed in Section 10.2. We consider their behaviour in such an extension:

a = b = y = 0	

r1 = load$_{rlx}$(a);
if (r1 == 0)
  store$_{sc}$(y,1)
else
  store$_{rlx}$(b,1)
	do
	  r2 = load$_{sc}$(y);
	  r3 = load$_{rlx}$(b)
	while (r2 + r3 == 0);
	store$_{rlx}$(a,1)

r1 = r3 = 1 $\wedge$ r2 = 0 BAD	

For r2 to be 0 when the loop exits, r3 must have been non-zero, and thus read from the store of 1 to y, so r1 must have read 0, so the outcome is forbidden.

a = b = x = y = 0		

r0 = load$_{sc}$(x);
if (r0 == 1)
  r1 = load$_{rlx}$(a)
else
  r1 = 0
if (r1 == 0)
  store$_{sc}$(y,1)
else
  store$_{rlx}$(b,1)
	do
	  r2 = load$_{sc}$(y);
	  r3 = load$_{rlx}$(b)
	while (r2 + r3 == 0);
	store$_{rlx}$(a,1)
		store$_{sc}$(x,1)

r0 = r1 = r3 = 1 $\wedge$ r2 = 0 BAD		

If r0 is 1, the program becomes the same as Causality test 14.

Causality test 16 The JMM intentionally does not guarantee coherence, and so Pugh allows this outcome.

x = 0	
r1 = load$_{rlx}$(x); store$_{rlx}$(x,1)	r2 = load$_{rlx}$(x); store$_{rlx}$(x,2)
r1 = 2 $\wedge$ r2 = 1 BAD	

In our semantics, which aims to enforce coherence (following C/C++11), none of the writes can be deordered with respect to the reads, as they are at the same location, so at least one of the writes has to read 0.

Causality test 17

x = y = 0	

r3 = load$_{rlx}$(x);
if (r3 != 42)
  store$_{rlx}$(x,42)
r1 = load$_{rlx}$(x);
store$_{rlx}$(y,r1)
	r2 = load$_{rlx}$(y);
	store$_{rlx}$(x,r2)

r1 = r2 = r3 = 42 OK	

In our memory model, Read-after-Write merging of the events corresponding to the r1 = load$_{rlx}$(x) into those corresponding to store$_{rlx}$(x,42) in the non-42 branches, and Read-after-Read merging into the one corresponding to r3 = load$_{rlx}$(x) in the 42 branch enables deordering the write of 42 to y.

Causality test 18

x = y = 0	

r3 = load$_{rlx}$(x);
if (r3 == 0)
  store$_{rlx}$(x,42)
r1 = load$_{rlx}$(x);
store$_{rlx}$(y,r1)
	r2 = load$_{rlx}$(y);
	store$_{rlx}$(x,r2)

r1 = r2 = r3 = 42 OK

Value-range speculation allows marking all reads of values different from 0 and 42 as dead, and thus this causality test behaves as causality test 17.

Causality test 19 This causality test and the next rely on a "join" construct (not in our syntax) that has the effect of *sequencing* the threads, but not of introducing synchronisation, so that writes from the later thread can be executed before the end of the earlier thread. This is a somewhat surprising notion of join, and is related to the issue discussed in Section 10.9.

x = y = 0		

join thread 3
r1 = load$_{rlx}$(x)
store$_{rlx}$(y,r1)
	r2 = load$_{rlx}$(y);
	store$_{rlx}$(x,r2)
		r3 = load$_{rlx}$(x);
		if (r3 != 42)
		  store$_{rlx}$(x,42)

r1 = r2 = r3 = 42 OK

We could model such a construct by adding a very ad-hoc "sequencing" transition that forces the execution of the joined thread to finish before the execution of the joining thread to resume. This litmus test would then behave like causality test 17, making this outcome allowed.

Causality test 20 Pugh allows this test, as, given the semantics of `join` above, it ought to behave like causality test 18.

x = y = 0		

join thread 3
r1 = load$_{rlx}$(x);
store$_{rlx}$(y,r1)
	r2 = load$_{rlx}$(y);
	store$_{rlx}$(x,r2)
		r3 = load$_{rlx}$(x);
		if (r3 == 0)
		  store$_{rlx}$(x,42)

r1 = r2 = r3 = 42 OK

Again, by extending our memory model with such a "sequencing" transition, we could obtain the same outcome.

6.4 Ševčík's litmus tests

Ševčík [74] describes a series of litmus tests that expose unsoundness of certain common optimisations in the Java memory model. They are mostly designed to highlight internal problems of the JMM; we discuss those that are still instructive.

Redundant Read after Read Elimination This is the JMM CSE trap litmus test on Page 46. The JMM does not allow this outcome, but Java compilers/JVMs still perform this optimisation. This shortcoming is one of the reasons the JMM was abandoned.

Irrelevant read introduction This litmus test tests whether introducing irrelevant (non-racy) reads changes the behaviour. In the JMM, this outcome is (mistakenly) forbidden, even though it becomes allowed if we introduce irrelevant read by uncommenting `r4 = load`$_{\texttt{rlx}}$`(x)`.

`x = y = z = 0`

`r1 = load`$_{\texttt{rlx}}$`(z);`	`store`$_{\texttt{rlx}}$`(x,1);`
`if (r1 == 0)`	`r2 = load`$_{\texttt{rlx}}$`(y)`
`  r3 = load`$_{\texttt{rlx}}$`(x);`	`store`$_{\texttt{rlx}}$`(z,r2)`
`  if (r3 == 1)`	
`    store`$_{\texttt{rlx}}$`(y,1)`	
`else`	
`  // r4 = load`$_{\texttt{rlx}}$`(x)`	
`  r4 = 1`	
`  store`$_{\texttt{rlx}}$`(y,r1)`	

`r1 = r2 = 1` OK

In our semantics, however, this outcome is possible in both cases: value-range speculation means reads need only consider values 0 and 1. The write of 1 to `x` by the second thread can be executed, and read from by the first thread, in both branches in the uncommented version. The program is then LB+po+data.

6.5 Miscellaneous tests of relaxed atomics

Coherence tests The "Co" family of litmus tests (CoRR0, CoRR1, CoRR2, CoWW, CoRW1, CoWR, CoRW, ... [50, §8]) tests for coherence (see Section 2.2). The side-conditions of deordering ensure coherence is enforced in the thread subsystem; the storage subsystem, following that of Power, also enforces coherence. For example, deordering of reads at the same location is forbidden, so the following litmus tests behaves as expected:

CoRR1

`x = 0`

`store`$_{\texttt{rlx}}$`(x,1)`	`r1 = load`$_{\texttt{rlx}}$`(x);`
	`r2 = load`$_{\texttt{rlx}}$`(x)`

`r1 = 1` $\wedge$ `r2 = 0` BAD

Random number generator This litmus test from folklore, a variation on LB+datas, is a trap litmus test for memory models based on sequences of executions, like the JMM. The straightforward execution where the first thread reads 0 from `x`, then writes 0 to `y`, and the second thread reads 0 from `y`, then writes 1 to `x`, could mistakenly be used to justify another execution where the first thread reads 1 from `x` from the previous execution, then writes 1 to `y`, so the second thread reads 1 from `y`, and thus writes 2 to `x`, and so on, allowing any value to be readable:

`x = y = 0`

`r1 = load`$_{\texttt{rlx}}$`(x);`	`r2 = load`$_{\texttt{rlx}}$`(y);`
`store`$_{\texttt{rlx}}$`(y,r1)`	`store`$_{\texttt{rlx}}$`(x,r2 + 1)`

`r1 = 42` $\wedge$ `r2 = 43` BAD

In our semantics, though, one of the two threads has to start by reading, and thus has to read the initial value, 0.

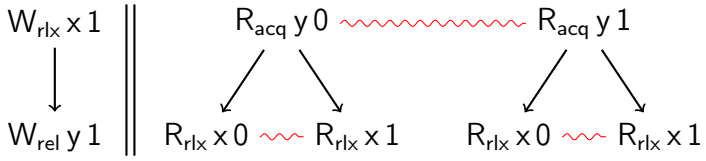
6.6 Synchronising accesses

Maranget et al. [50] describe a few common synchronisation patterns. We consider versions with various C/C++11 memory orders.

MP The MP+rlx-rel+acq-rlx litmus test illustrates that the release and acquire memory orders implement message passing. A thread reading the flag `y` as set with an acquire read has to see the data written to `x` when the flag has been set with a release write, and cannot see stale data:

x = y = 0	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$ $\texttt{store}_{\texttt{rel}}\texttt{(y,1)}$	$\texttt{r1 = load}_{\texttt{acq}}\texttt{(y);}$ $\texttt{r2 = load}_{\texttt{rlx}}\texttt{(x)}$
$\texttt{r1 = 1} \wedge \texttt{r2 = 0}$ BAD	

As the write of the flag is a release write, the thread subsystem forbids deordering, and thus enforces it is executed after the first; the barrier in the storage subsystem enforces it is propagated after the write of the data. As the read of the flag is an acquire read, the thread subsystem forbids deordering, and thus enforces it is executed before the read of the data; if it reads 1, then the write of the data has to have propagated, so the read of the data reads 1.



If the write to the flag `y` is changed to be relaxed, then the write to the data `x` is not forced to propagate to the second thread before the write to the flag, and the outcome is allowed.

WRC For the message passing idiom to be truly usable, it needs to work when iterated: a first thread writes the data, and flags it as ready, a second thread sees the flag, and passes this information through another flag, and so on; the final thread reading the final flag should then the see the data. This is called Write-to-Read Causality by Boehm and Adve [20], and illustrated by the following litmus test:

WRC+rlx-rel+acq-rel+acq-rlx

x = y = 0		
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$ $\texttt{store}_{\texttt{rel}}\texttt{(y,1)}$	$\texttt{r1 = load}_{\texttt{acq}}\texttt{(y);}$ $\texttt{store}_{\texttt{rel}}\texttt{(z,r1)}$	$\texttt{r2 = load}_{\texttt{acq}}\texttt{(z);}$ $\texttt{r3 = load}_{\texttt{rlx}}\texttt{(x)}$
$\texttt{r2 = 1} \wedge \texttt{r3 = 0}$ BAD		

The acquire read and the release write of the middle thread cannot be reordered (that is the point of acquire reads and release writes), and `lwsync` barriers are "cumulative" [72], enforcing the desired behaviour in the storage subsystem.

Release sequences Again to ensure the message passing idiom is truly usable, reading from a coherence-successor of a release write should have the same effect as reading from the release write itself. C/C++11 formalises this with "release sequences". For example, in the program below, if the relaxed write of 2 to `y` is after the release write of 1 to `y` in coherence (enforced by `y = 2`), then reading 2 from `y` with an acquire read ensures we will read 1 for `x`.

`x = y = 0`	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$	$\texttt{r1 = load}_{\texttt{acq}}\texttt{(y);}$
$\texttt{store}_{\texttt{rel}}\texttt{(y,1);}$	$\texttt{r2 = load}_{\texttt{rlx}}\texttt{(x)}$
$\texttt{store}_{\texttt{rlx}}\texttt{(y,2)}$	

$\texttt{r1 = 2} \wedge \texttt{r2 = 0}$ BAD

In our case, this follows from the treatment of release writes and coherence by the storage subsystem.

SB Some synchronisation patterns, like Dekker's algorithm, rely on sequential consistency for the accesses used to implement them, and in particular on the fact that the propagation of stores is not delayed by buffers. A litmus test that illustrates store buffering is SB:

SB+rlx-sc+rlx-sc

`x = y = 0`	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$	$\texttt{store}_{\texttt{rlx}}\texttt{(y,1);}$
$\texttt{r1 = load}_{\texttt{sc}}\texttt{(y)}$	$\texttt{r2 = load}_{\texttt{sc}}\texttt{(x)}$

$\texttt{r1 = r2 = 0}$ BAD

The `sync` that forms the first part of each read ensures that the thread's write has propagated before the read is executed, so that at least one of the threads has to read 1.

This behaviour of SC accesses is stricter than in C/C++11, and prevents us from using the same compilation scheme to TSO.

IRIW SC reads ensure that independent reads of independent writes are seen in the same order:

IRIW+rlx+rlx+rlx-sc+rlx-sc

`x = y = 0`			
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(y,1)}$	$\texttt{r1 = load}_{\texttt{rlx}}\texttt{(x);}$	$\texttt{r3 = load}_{\texttt{rlx}}\texttt{(y);}$
		$\texttt{r2 = load}_{\texttt{sc}}\texttt{(y)}$	$\texttt{r4 = load}_{\texttt{sc}}\texttt{(x)}$

$\texttt{r1 = 1} \wedge \texttt{r2 = 0} \wedge \texttt{r3 = 0} \wedge \texttt{r4 = 1}$ BAD

If the accesses are relaxed, as in the examples in Section 2.2, the relaxed behaviour is allowed by the storage subsystem, in the same way it is allowed in Power: the write to `x` can propagate to the third thread but not to the fourth, and the write to `y` to the fourth but no the third at the time the reads are executed. Moreover, deordering of reads is also enough to expose this outcome (as in the Power thread subsystem).

RWC Moving one of the writes of SB out to another thread yields the RWC litmus test of Boehm and Adve [20]:

RWC+rlx+rlx-sc+rlx-sc

x = y = 0		
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$	$\texttt{r1 = load}_{\texttt{rlx}}\texttt{(x);}$ $\texttt{r2 = load}_{\texttt{sc}}\texttt{(y)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(y,1);}$ $\texttt{r3 = load}_{\texttt{sc}}\texttt{(x)}$

$\texttt{r1 = r2 = 1} \wedge \texttt{r3 = 0}$ BAD

S This variant of the S litmus test tests the effect of control dependencies from reads to writes.

S+rlx-sc+ctrl

x = y = 0	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,2);}$ $\texttt{store}_{\texttt{sc}}\texttt{(y,1)}$	$\texttt{r1 = load}_{\texttt{rlx}}\texttt{(y);}$ `if (r1 == 1)` $\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$

$\texttt{x = 2} \wedge \texttt{r1 = 1}$ BAD

Relaxing the SC store, or removing the control dependency allows the relaxed outcome.

R This litmus test checks whether the union of coherence at different locations forces a write by one thread to be visible by another. It is similar to MP, with coherence replacing the reading of the flag.

R+rlx-sc+rlx-sc

x = y = 0	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$ $\texttt{store}_{\texttt{sc}}\texttt{(y,1)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(y,2);}$ $\texttt{r1 = load}_{\texttt{sc}}\texttt{(x)}$

$\texttt{y = 2} \wedge \texttt{r1 = 0}$ BAD

If the accesses are relaxed, the writes of the first thread can be deordered, and so can the read and the write of the second thread.

2+2W This litmus test checks whether release writes are enough to prevent cycles in the union of coherence at different locations.

2+2W+rlx-rel+rlx-rel

x = y = 0	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$ $\texttt{store}_{\texttt{rel}}\texttt{(y,2)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(y,1);}$ $\texttt{store}_{\texttt{rel}}\texttt{(x,2)}$

$\texttt{x = y = 1}$ BAD

Making the second write of each thread release prevents the writes from being deordered (in the thread semantics) and propagated out of order (in the storage subsystem), which prevents the program from executing "backwards".

Mutual exclusion This litmus test checks whether RMWs actually implement mutual exclusion, by checking whether two RMWs can both succeed (setting the first register argument to 1) even though they are both trying to read the same value (0; the second register argument is set to the value read), and both make it unavailable:

mutual exclusion

x = y = 0	
$\texttt{r1, r2 = rmw}_{\texttt{acqrel/acq}}\texttt{(x,0,1)}$	$\texttt{r3, r4 = rmw}_{\texttt{acqrel/acq}}\texttt{(x,0,2)}$

$\texttt{r}_1 = \texttt{r}_3 = 1 \wedge \texttt{r}_2 = \texttt{r}_4 = 0$ BAD

6.7 Undefined behaviour

6.7.1 LB and undefined behaviour

The undefined-behaviour example of Batty et al. [12, §7] can be expressed in our setting as the program below, where `a` represents the array in the original example:

Undefined behaviour (Batty et al.)

`x = y = a = 0`

`r1 = load`$_{\texttt{rlx}}$`(x);`	`r2 = load`$_{\texttt{rlx}}$`(y);`
`if (r1 == 0)`	`store`$_{\texttt{rlx}}$`(x,r2)`
`  r3 = load`$_{\texttt{rlx}}$`(a)`	
`else`	
`  undef`	
`store`$_{\texttt{rlx}}$`(y,1)`	

`undef` OK

In this example, the order of the conditional and the write to `y` should not matter, because they concern different locations. Therefore, this program should behave the same as

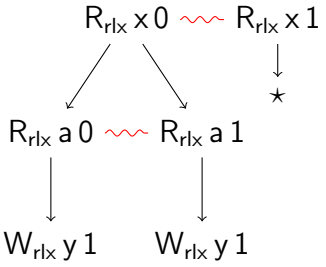
Undefined behaviour, reordered (Batty et al.)

`x = y = a = 0`

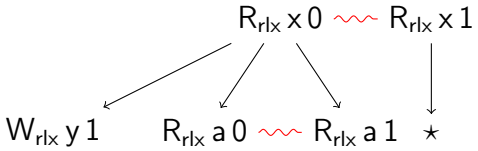
`store`$_{\texttt{rlx}}$`(y,1)`	`r2 = load`$_{\texttt{rlx}}$`(y);`
`r1 = load`$_{\texttt{rlx}}$`(x);`	`store`$_{\texttt{rlx}}$`(x,r2)`
`if (r1 == 0)`	
`  r3 = load`$_{\texttt{rlx}}$`(a)`	
`else`	
`  undef`	

By executing this program in a sequentially consistent way, we can show that the undefined behaviour can be triggered. The first thread writes 1 to `y`, the second thread reads it, and writes 1 to `x`, which the first thread reads into `r1`. Therefore, the undefined behaviour is possible.

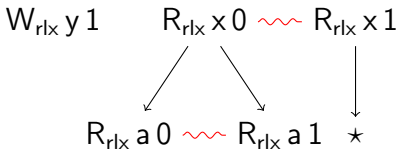
In our memory model, the initial event structure of the first thread of the first example is the following:



It can be deordered to



which can itself be deordered to:



as the $\star$ acts as a joker for deordering.
The write of 1 to `y` can then be executed (and propagated) for the second thread to read it, and write 1 to `x`, which can then be read, which allows executing $\star$.

The order of the conditional and the write in the source does not matter, as desired.

6.7.2 Undefined behaviour does not trigger itself

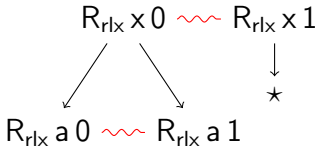
On the other hand, if the write to `y` is removed, as in the example below, the presence of a source of undefined behaviour in the program should not be enough for it to trigger itself.

`x = y = a = 0`

`r1 = load`$_{rlx}$`(x);`	`r2 = load`$_{rlx}$`(y);`
`if (r1 == 0)`	`store`$_{rlx}$`(x,r2)`
`  r3 = load`$_{rlx}$`(a)`	
`else`	
`  undef`	

`undef` BAD

If we remove the write to `y`, in our memory model, the initial event structure of the first thread is now



The write of 1 to `y` cannot be deordered anymore, so the undefined behaviour cannot be triggered any more, as desired.

Chapter 7

Implementability

We show that our memory model is implementable on x86-TSO and Power. Following Batty et al. [13], rather than showing the correctness of any specific, entire compiler, we instead show the correctness of the expected compilation schemes, which propose a sequence of instructions to implement each kind of memory access (assuming that the rest of the compilation preserves memory accesses).

Our correctness criterion is the preservation of final values of locations that are syntactically accessed only by a single thread. This notion is easy to define not only for our memory model, but also for Power and x86-TSO.

Definition 69. A *behaviour* is a map from locations to values.

Definition 70. The *behaviours of a program P* is the set of such maps, the domains of which is the set of locations that are syntactically accessed only by a single thread, and such that

- there exists an execution such that for each such location, the associated value is
  - in x86-TSO, after all threads have finished, and all buffers are flushed, the value in the main memory;
  - in Power, after all threads have finished, and the storage subsystem has finished, the coherence-final value; and
  - in our memory model, after all threads have finished, and the storage subsystem has finished, the coherence-final value;
- or the program exhibits undefined behaviour.

The definition for our memory model is the same as for Power, except that in our memory model, some program-order-final writes might be merged, but then they are of the same value.
The undefined behaviour can come from a race, or from a thread-local trigger of undefined behaviour.

We can extend the notion of behaviour to include registers, by adding a write of each register to a distinguished location at the end of each thread.

Divergence In the hardware memory models, locks are implemented by a loop around an RMW, which can fail by diverging. In our semantics, this is represented by the unsuccessful locking events. This causes a mismatch in style that we circumvent; we discuss this mismatch in Section 10.6.

7.1 Implementability on x86-TSO

The compilation scheme on x86-TSO of Terekhov [80] is as follows:

operation	x86-TSO implementation
non-sc load	`mov`
sc load	`mfence; mov`
non-sc store	`mov`
sc store	`mov; mfence`
lock	`lock cmpxch; test; jnz`
unlock	`mov`
RMW	`lock cmpxch`

The `test; jnz` represents (in our setting) a `repeat` around the `cmpxch`.

For C/C++11, where the behaviour of SC accesses is weaker, it is possible to use an amended compilation scheme where an SC load is implemented without an `mfence` [77].

Theorem 1 (x86-TSO compilation scheme soundness)**.** *All the behaviours of the compilation to x86-TSO of a program according to the compilation scheme are behaviours of the original.*

Proof. We show a tighter, more intensional correspondence by establishing a simulation between TSO executions and executions in our memory model. To do this, we need to know the sources of instructions, for example whether a read in the TSO execution comes from a load or an RMW in our source program. This can be done by instrumenting the TSO model and the compilation scheme with ghost state.

We can then establish a simulation by encoding the TSO state in our memory model. The state of the thread itself, without the buffer, is simulated by the corresponding event structure according to Definition 43. As TSO executes threads in order, there is no deordering. The TSO buffer of a thread is simulated by a storage subsystem such that, for each non-SC write in the TSO buffer of that thread, there is a write from that thread that has not propagated to the other threads, and no other unpropagated writes from that thread; moreover, if a write is in a buffer before another write at the same location, then they are coherence-committed in that order, and all non-propagated writes from that thread are coherence-committed after all propagated writes. Because of the trailing `mfence`, there can only be one SC write in the TSO buffer of a thread, in which case it is the last write of the buffer. The TSO main memory is simulated by a storage subsystem such that all propagated writes at a given location are totally ordered by coherence, and such that the value of the last such write is the value associated to that location by the TSO main memory. The storage subsystem needs to simulate all the TSO buffers, and the TSO main memory.

Then, each step of the execution (which takes place in program order) of the compiled program on TSO can be simulated by a series of steps of the original program in our memory model:

- a read `mov` is matched by a read being executed; for an SC read, this requires executing the leading `sync`, which requires all writes by that thread to be propagated, which is ensured by the leading `mfence`; because of the `mfence` after an SC write, no thread ever reads from an SC write in its buffer, always from main memory, so the right value can be read;

- a write `mov` for a non-SC write is matched by the write being issued to the storage subsystem;

- a write flushing is matched by propagating the write to all the threads, and coherence commitments to make it the latest write past coherence point at that location; moreover, for an SC write, this is matched by first executing the write, which requires executing the leading `sync`, which requires all previous writes by that thread to be propagated, which is ensured by the first-in first-out nature of TSO buffers;

- an `mfence`: nothing, but it requires all writes by that thread to be propagated; see the read and write cases;

- an unlock `mov` is matched by the unlock, that is, an `lwsync` and a write, being issued to the storage subsystem;

- a `lock cmpxch`: depending on its source:

  - from a RMW: an RMW (which might fail);
  - from a lock:
    * if the lock succeeds now: a successful locking;
    * if the lock succeeds later: nothing;
    * if the lock never succeeds (and the TSO execution diverges): an unsuccessful locking. □

7.2 Implementability on Power

One of the design goals of the memory model is to be cheaply implementable on Power, by mapping relaxed and nonatomic reads and writes to plain assembly loads and stores, without memory barriers or any other synchronisation. We show that the (amended, as given by Sewell [77]) Power compilation scheme for C/C++ of McKenney and Silvera [56, 13] can be used for our memory model:

operation	Power implementation
nonatomic load	`ld`
load relaxed	`ld`
load acquire	`ld; cmp; bc; isync`
load seq cst	`sync; ld; cmp; bc; isync`
nonatomic store	`st`
store relaxed	`st`
store release	`lwsync; st`
store seq cst	`sync; st`
lock	`lwarx; cmp; bc;` `stwcx; bc; isync`
unlock	`lwsync; st`
RMW	`lwsync; lwarx; cmp; bc;` `stwcx; bc; isync`

7.2.1 Overview

We first argue informally why the different features of the Power memory model are accounted for in our memory model.

Dependencies Power allows arbitrary thread-local reordering of accesses to different locations (without dependencies or barriers), which is what makes the basic two-thread two-location two-event litmus tests (MP, SB, LB, R, S, 2+2W, see Section 6) of Sarkar et al. [72] all permitted there. Our semantics allows all those reorderings, using the first and second rules of Section 4.4.7 for R,W→W and R,W→R respectively.

Our semantics has to ensure that semantic dependencies (data, address, and control) from a read to a write are respected, otherwise the thin-air examples become permitted. The syntactic dependencies that Power respects over-approximate these. For example, take the first thread of LB+datas, on Page 77. This will be mapped, by the compilation scheme for relaxed atomics, to a Power load and store instruction

```
ld r1,0(rx) // load doubleword from [rx] to r1
std r1,0(ry) // store doubleword in r1 to [ry]
```

where `rx` and `ry` are registers holding the addresses of `x` and `y`. In the Power semantics, that store cannot become visible to other threads until its address and data are known (ultimately, because the architecture does not permit observable value speculation), validating the fact that our semantics does not allow deordering of the write w.r.t. the read.

In contrast, Power does not respect control dependencies from a read to a read (from a different location), as the hardware can and does speculate conditional branches. For example,

```
r1 = load_rlx(x);
if (r1 == 1)
  r2 = load_rlx(y)
```

will be mapped to

```
  ld r1,0(rx) // load doubleword from [rx] to r1
  cmpdi r1,1  // compare r1 with 1
  bne .label  // branch if nonequal, to label
  ld r2,0(ry) // load doubleword from [ry] to r2
.label
```

In Power, the second load can be satisfied before the first, and this is accommodated in our semantics by permitting the second read (`r2 = load_rlx(y)`) to be deordered with respect to the first.

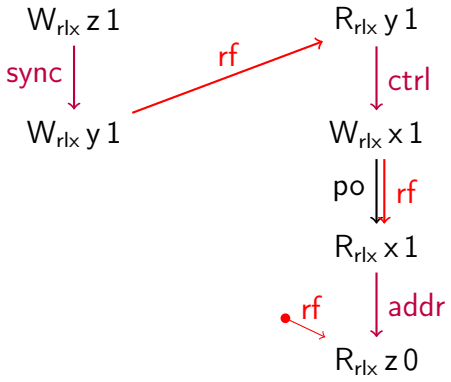
Non-multi-copy-atomicity Power also has non-multi-copy-atomic write propagation, allowing a write to propagating to other threads one-by-one. Our semantics accommodates that with a storage subsystem model that does exactly the same.

Write forwarding Power allows reads to read from program-order-before writes thread-locally, without going through the storage subsystem, as the PPOCA litmus test illustrates [72]:

PPOCA

x = y = 0	
store(z,1);	r1 = load(y); // 1
sync;	if (r1 == 1)
store(y,1)	store(x,1);
	r2 = load(x); // 1
	r3 = load(z+r2-r2) // 0

r1 = r2 = 1 ∧ r3 = 0 OK



Test PPOCA: Allowed

This is just an instance of Read after Write, and is covered by merging in our semantics.

Registers An important source of relaxed behaviour in Power has to do with the fact that operations on registers do not enforce program order, because of register shadowing, as illustrated by litmus tests MP+sync+rs (also called MP+dmb/sync+rs) and LB+rs [72]:

MP+sync+rs

x = y = 0	
store(x,1);	r1 = load(y); // 1
sync;	r1 = r3;
store(y,1)	r3 = load(x) // 0

r1 = 1 ∧ r3 = 0 OK

LB+rs

x = y = 0	
r1 = load(x); // 2	r3 = load(y); // 1
r2 = r1; // 2	r3 = r3 + 1; // 2
r1 = 1;	store(x,r3)
store(y,r1)	

r1 = 1 ∧ r2 = 2 ∧ r3 = 2 ∧ y = 1 ∧ x = 2 OK

In our semantics, registers are resolved to concrete values when building the event structure, so the program order of register accesses does not impose any order on execution.

Locks The crucial property of locks is that they enforce order between actions before an unlock, and actions after the next lock. In our memory model, this is enforced by preventing reordering with respect to locks, and deordering of unlocks with respect to other actions. In the standard Power lock implementation [38], as analysed by Sarkar et al. [71, §], (1) the `lwsync` preceding the write signalling the unlocking enforces the order between the actions programs-before the unlock and the unlock itself; (2) the branching on the result of the store-conditional, followed by an `isync` in the implementation of `lock`, enforces the order between the lock itself and the actions program-order-after it; and (3) the load-reserve/store-conditional pair enforces the order between the unlock and the lock.

Acquire An acquire read is implemented by a load followed by an artificial branching followed by an `isync`. This sequence of instructions prevents the execution of instructions that are after the acquire read (in program order) before the acquire read itself (in execution order), which matches the constraint in our model that memory actions cannot be deordered with respect to acquires.

Release A release write is implemented by an `lwsync` followed by a store. The `lwsync` can only be committed once all previous memory accesses are committed, which matches the constraint that release writes cannot be deordered. The effect of the `lwsync` in the storage subsystem is the same, by construction.

SC An SC read is implemented by a `sync` followed by a load, and an SC write by a `sync` followed by a store. The `sync` can only be committed once all previous memory accesses are committed, and memory actions after the `sync` can only be committed once the `sync` has been acknowledged, which matches the constraint that SC accesses cannot be deordered, and memory actions cannot be deordered with respect to SC accesses, and that the thread is suspended until the `sync` is acknowledged. The effect of the `sync` in the storage subsystem is the same, by construction.

RMW An RMW is implemented by an initial `lwsync`, to have the strength of a release, followed by a load-reserve, a store-conditional, and a final artificial `cmp; bc; isync`, to have the strength of an acquire. It is similar to how locks are implemented, but without the loop that ensures the success of the store-conditional of the lock.

Undef By definition, executing `undef` counts as having any behaviour, so there is nothing to show.

7.2.2 Details

To show the implementability of our memory model on Power more precisely, we rely on a few properties of the Power memory model:

Property 1 (Restarting can be removed)**.** *For each Power execution (with restarts), there is subsequence of that execution that does not have any restarts and is also an execution with the same behaviours.*

Property 2 (Speculative writes can be removed)**.** *For each Power execution, there is a subsequence of that execution that does not have any initiated but non-committed writes, and is also an execution with the same behaviours.*

Property 3 (Speculative reads can be removed)**.** *For each Power execution, there is a subsequence of that execution that does not have any satisfied but non-committed reads, and is also an execution with the same behaviours.*

We assume that these properties hold. Restarted instructions, non-committed writes, and writes depending on non-committed reads do not affect the storage subsystem. To prove they hold, it would thus suffice to show that instructions that are actually executed by the thread subsystem do not depend on the partial execution of other instructions, by careful inspection of the thread subsystem semantics.

In Power, some events that correspond to memory actions (for example read satisfactions or write initiations, as opposed to register operations) can appear in the execution even though they do not correspond to an event of the event structure. For example, in the following program sketch, the write of 1 to `y` and the read of `z` can be initiated and satisfied, respectively, even though neither appear in the event structure, because they are inside an impossible `if`:

```
r1 = x;
if (false)
  y = 1;
  r2 = z
```

However, neither can be committed, so they can be removed.

To make this notion precise, we could (but will not do explicitly so here, as this would be lengthy, but routine) instrument the computation of the event structure of a program to annotate each event with the instruction it came from.

Definition 71. A Power event is *orphaned* when there is no event in the annotated event structure of the program that is annotated with its instruction instance.

Lemma 4 (Power orphaned events can be removed)**.** *For each Power execution, there is a subsequence of that execution that does not have any orphaned events, and is also an execution with the same behaviours.*

Proof. If an event does not appear in the event structure, then it depends on instructions (branches, etc.) that cannot be committed no matter what values are read, so the event cannot be committed, so it cannot affect the execution (see Properties 3 and 2). □

Theorem 2 (Power compilation scheme soundness)**.** *All the behaviours of the compilation to Power of a program according to the compilation scheme are behaviours of the original.*

Proof sketch. As for TSO, we establish a simulation. Again, we instrument the model and the compilation scheme to know the source of instructions. We now also need to look ahead in the trace to know whether some operations are restarted.
The state of the storage subsystem in our memory model is the same as that of Power, so we do not have to consider internal storage subsystem transitions of Power, only transitions involving the thread subsystem.
Moreover, we can rely on Properties 1, 2, and 3, and Lemma 4 to simplify the Power execution.

– fetch instruction: nothing. All non-orphaned instructions are, in a sense, already fetched; and, by Lemma 4, orphaned instructions do not have to be considered.

- commit in-flight instruction:
  - for thread-local instructions:
    - for register operations: nothing: these operations are resolved in the construction of the event structure.
    - for a `cmp` or a `bc` (from an `if`, an acquire read, a lock, or an RMW): nothing: these operations are resolved in the construction of the event structure.
  - for a read: nothing (see satisfaction).
  - for a write:
    - from a store (plain): deorder and execute the write. As the write is committed, previous barriers and instructions at the same location have been committed, so this deordering can be performed. If this write is a release write, the lwsync is also issued to the storage subsystem in our memory model.
    - from an unlock: nothing; it is done with the lwsync.
    - write-exclusive:
      - coming from a lock:
        - if the lock succeeds now: successfully execute the lock, which is possible, as the state of the storage subsystem is the same.
        - if the lock succeeds later: nothing: only the successful loop iteration will be mirrored.
        - if the lock never succeeds: execute an unsuccessful lock.
      - coming from an RMW:
        - if it succeeds: execute the RMW; the result will be the same, as the state of the storage subsystem is the same.
        - if it fails: nothing: it was already done when reading.
  - for a sync: issue the sync.
  - for an lwsync:
    - from a plain write: it is from a release write; nothing: it is done with the release write.
    - from an RMW: nothing: it is done with the write.
    - from an unlock: execute the unlock (which contains an lwsync and a write).
  - for an isync:
    - from a plain load: it is from an acquire load: nothing: it is implicit in a read acquire.
    - from a lock/RMW: nothing: it is implicit.
- accept sync barrier acknowledgement: resume thread.
- satisfy memory read from storage subsystem:
  - if the read does not appear in the event structure: nothing (see Lemma 4).
  - if the read appears in the event structure:
    - for a read from a (plain) load:

- if the read will be restarted: nothing (see Property 1).
- if the read will not be restarted:
  First, we know that all program-order-previous syncs are committed and acknowledged, and all program-order-previous isyncs are committed.
  Second, the read is not program-order-after another non-committed read at the same location that reads from a different write, as, if it were the case, this read would be restarted.
  Third, the read is not program-order-after a non-committed write at the same location, as, if it were the case, this read would be restarted: it would only be kept if a write were forwarded to it (see that case).
  Now, if this read can read from the same write as the non-committed program-order-before reads, then this read, and the non-committed program-order-before reads, can be deordered and executed now. As the state of the storage subsystem is the same, the same value can be read.

- for a read from a load reserve from a lock/RMW:
  - if the store exclusive succeeds: nothing: it will be done at time of write-exclusive.
  - if the store exclusive fails: execute the (failed) RMW now; the state of the storage subsystem is the same, so the same value can be read.

- satisfy memory read by forwarding an in-flight write directly to reading instruction:
  - if the read does not appear in the event structure: nothing (see Lemma 4).
  - if the read appears in the event structure:
    - for a read from a (plain) load: As there is no program-order-intervening memory write that might be to the same address, then there is no intervening memory write to the same address in the event structure. As all program-order-previous syncs are committed and acknowledged and all program-order-previous isyncs are committed, then there is no intervening acquire action. Therefore, a Read-after-Write backward merging step can be taken.
    - for a read from a lock/RMW: see the corresponding case from "satisfy memory read from storage subsystem".

- register read from previous write: nothing: this is resolved in the construction of the event structure.

- register read from initial register state: nothing: this is resolved in the construction of the event structure.

- internal computation step: nothing: this is resolved in the construction of the event structure. □

7.3 Remarks

As expected, because these are hardware memory models, we do not need to use speculation steps, which were introduced to model compiler optimisations.

7.4 Related work

Batty et al. [15] show that C/C++11 is implementable on x86-TSO using the (amended) mapping of Terekhov. Sarkar et al. [71] and Batty et al. [13], as amended by Lahav et al. [44], show that C/C++11 is implementable on Power using the (corrected) mapping of McKenney and Silvera. Flur et al. [32] show that C/C++11 is implementable on ARMv8.

Because of the axiomatic style of C/C++11, the style of these proofs is quite different: they build C/C++11 candidate executions based on the traces of the hardware memory models, and show that they fulfil the consistency predicate.

Chapter 8

Soundness of optimisations

To illustrate that our memory model forms an envelope around reasonable compiler optimisations, we show that some *syntactic* compiler optimisations, that is, changes of the text of the program, are indeed accounted for, in terms of behaviour, by our memory model.

Conventionally, a compiler optimisation is a function from some text to text (or intermediate language to intermediate language, in practice). Here, we just focus on single points of such a function. We take advantage of the fact that our programs do not feature divergence to simplify our definitions.

Definition 72. A *program optimisation* (respectively *thread optimisation*) is a pair of programs (threads), called the initial program (thread), and the resulting program (thread).

Given such an optimisation, we can then ask whether it actually is an optimisation according to our memory model:

Definition 73. A program optimisation is *sound* when the behaviours of the resulting program are behaviours of the initial program.

Definition 74. A thread optimisation (`T`,`T'`) is *sound* when, for all list of threads `Ts`, the behaviours of `T'` running concurrently with `Ts` are behaviours of `T` running concurrently with `Ts`.

In most of the examples below, we will focus on thread optimisations.

8.1 The interaction of value-range analysis and merging

The usual method to show the correctness of a thread optimisation is to show that the resulting thread can simulate the original thread. However, value-range analysis makes this kind of reasoning unsound, as value-range analysis allows the other threads to take advantage of the fact that the resulting thread *cannot* simulate some of the behaviour of the original to deorder more aggressively.

An example of value-range speculation taking advantage of the fact that a thread cannot do something is the following program: if the first write to `x` of the second thread, which writes 1, is optimised away, then value-range analysis can determine that the read of `x` by the first thread cannot read 1, and the first thread can therefore reorder the store of 1 to `y` with its read from `x`.

x = y = 0

r1 = load$_{rlx}$(x);	store$_{rlx}$(x,1);	r2 = load$_{rlx}$(y);
if (r1 != 1)	store$_{rlx}$(x,2)	if (r2 == 1)
store$_{rlx}$(y,1)		store$_{rlx}$(x,3)

r1 = 3 OK

This is accounted for by our memory model by merging: if the first write to x is merged into the second, then the write to y can be executed before the read of x, which means the write of 3 to x can be executed before the read of x, which can therefore read from it.

However, this is made possible by the merging steps of the semantics, which only allow merging of non-synchronising accesses. Replacing the two RMWs by a single one in the following program is unsound: without the optimisation, the second thread cannot speculate that it will not read 1 for y.

Merging of RMWs

x = y = 0

r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
r3, r4 = rmw$_{acqrel/acq}$(y,0,1);	if (r2 != 1)
r5, r6 = rmw$_{acqrel/acq}$(y,1,2)	store$_{rlx}$(x,2)

r1 = r2 = 2 BAD

With the optimisation, it can, and this enables the r1 = r2 = 2 outcome:

Merging of RMWs (merged)

x = y = 0

r1 = load$_{rlx}$(x);	r2 = load$_{rlx}$(y);
r5, r6 = rmw$_{acqrel/acq}$(y,0,2)	if (r2 != 1)
r3 = r5;	store$_{rlx}$(x,2)
r4 = r5	

r1 = r2 = 2 OK

This example can be adapted to release writes: in the program below, the write of 1 makes it impossible for the second thread to exclude the possibility it can read 1, and so it cannot deorder the write to x and execute it early (the read of x is an acquire read to prevent reordering in the first thread, and the writes to y are release writes to prevent merging). However, if the write of 1 is optimised away, then the behaviour is allowed.

OWE-on-release

x = y = 0

r1 = load$_{acq}$(x);	r2 = load$_{rlx}$(y);
store$_{rel}$(y,1);	if (r2 != 1)
store$_{rel}$(y,2)	store$_{rlx}$(x,1)

r1 = 1 $\wedge$ r2 = 2 forbidden

This shows that value-range analysis makes seemingly benign optimisations on synchronising accesses unsound. We return to this in Section 10.10. The optimisations we consider in this chapter are optimisations of non-synchronising accesses.

8.2 Auxiliary definitions for syntactic program manipulation

Definition 75. *Contexts*, and *list of statements contexts*, are defined mutually inductively by

`C`	::=	statement context
	| `[]`	hole
	| `if (`e_1 `cmp` e_2`) Cs else ss`	left conditional
	| `if (`e_1 `cmp` e_2`) ss else Cs`	right conditional
`Cs`	::=	list of statements context
	| s_1`; ...;` s_m`; C;` s_{m+1}`; ...;` s_n	sequential composition context
	| `{ Cs }`	block context

We denote *concatenation* with '`;`', and identify `C` and `Cs` when interchangeable.

Definition 76. *Plugging* statements `ss` in a context `C` (respectively `Cs`), `C[ss]` (respectively `Cs[ss]`), is defined recursively on the context:

- case `[]`: `ss`;
- case `if (`e_1 `cmp` e_2`) Cs else ss'`:
  `if (`e_1 `cmp` e_2`) Cs[ss] else ss'`
- ...
- case s_1`; ...;` s_m`; C;` s_{m+1}`; ...;` s_n:
  s_1`; ...;` s_m`; C[ss];` s_{m+1}`; ...;` s_n
- ...

Definition 77. The *left* of a context `C` (respectively `Cs`), left(`Cs`) (respectively left(`Cs`)), is defined recursively on the context:

- case `[]`: empty list of statements
- case `if (`e_1 `cmp` e_2`) Cs else ss`: left(`Cs`);
- ...
- s_1 `; ...;` s_m`; C;` s_{m+1}`; ...;` s_n :
  s_1 `; ...;` s_m`;` left(`C`);
- ...

Definition 78. A statement *accesses a location* `x` when it is of the form `C[r = load`$_{\texttt{rmo}}$`(x)]`, `C[store`$_{\texttt{wmo}}$`(x,e)]`, or `C[r1, r2 = rmw`$_{\texttt{acqrel/acq}}$`(x,`e_1`,`e_2`)]`.

Definition 79. A statement *writes to register* `r` when it is of the form `C[r = load`$_{\texttt{rmo}}$`(x)]`, `C[r = e]`, or `C[r1, r2 = rmw`$_{\texttt{acqrel/acq}}$`(x,`e_1`,`e_2`)]`.

8.3 Register operations

Because the operational semantics works with the event structures, and not the source program, any optimisation that preserves event structures is sound:

Lemma 5. *If* $[\![ss_1]\!]_{\mu_0} = [\![ss_2]\!]_{\mu_0}$*, where* μ_0 *is the initial register state, then changing* ss_1 *to* ss_2 *is a sound optimisation.*

Proof. The semantics only considers the code of the threads through their event structures. □

This abstracts the part of the work (and the complexity) of a compiler that has to do with rearranging register operations, and selecting the right instructions to implement them. For example, a compiler can perform strength reduction, changing `2 * e` to the faster `e << 1` (assuming our language is extended with C/C++11-like multiplication and bitwise operators). This does not change the event structures, so it behaves the same in our memory model.

This works more generally if the first event structure can transition to the second:

Lemma 6. *If* $[\![ss_1]\!]_{\mu_0}$ *can transition to* $[\![ss_2]\!]_{\mu_0}$ *with silent, non-VRS transitions, then changing* ss_1 *to* ss_2 *is a sound optimisation.*

Proof. If the former can transition to the latter with silent, non-VRS transitions, then the behaviour of the latter is behaviour of the former, as witnessed by the concatenation of the traces. □

For example, the event structures for LB+ctrldata+po (Page 16) are the same as those for LB+ctrldata+ctrl-double (Page 16), so they have the same behaviour.

8.4 Common subexpression elimination

Our memory model captures common subexpression elimination with merging steps. In a language like C/C++11, common subexpression elimination would change the following expression, which reads `x` and `y` twice, and computes their sum twice:

```
r5 = (x + y) + (x + y)
```

into the following expression, that does that only once:

```
r6 = x + y;
r5 = r6 + r6
```

(where `r6` is not used by the rest of the program). In our setting, which, like compiler intermediate languages, makes memory accesses explicit, the former expression would have to be split up into the following statements (where `r1` through `r4` are not used by the rest of the program):

```
r1 = load_na(x);
r2 = load_na(y);
r3 = load_na(x);
r4 = load_na(y);
r5 = (r1 + r2) + (r3 + r4)
```

This separates the memory aspect from the thread-local, register aspect (as discussed in Section 2.1). As we show below, optimising as follows, by replacing memory accesses by register operations, is sound:

```
r1 = load_na(x);
r2 = load_na(y);
r3 = r1;
r4 = r2;
r5 = (r1 + r2) + (r3 + r4)
```

Now that we have exposed that `r3` and `r1` contain the same value, and similarly `r4` and `r2`, optimising the register operations follows from Lemma 5, and the result of the optimisation corresponds to the optimised version above:

```
r1 = load_rlx(x);
r2 = load_rlx(y);
r6 = r1 + r2;
r5 = r6 + r6
```

We can now express the effect of common subexpression elimination on memory accesses:

Theorem 3. *If* left(Cs_2) *does not access* `x` *nor write to* `r1`*, does not contain any SC accesses, and*

- *either* $mo1 = mo2 = na$*, and there is no release action followed by an acquire action in* left(Cs_2)*;*
- *or* $mo1 \in \{rlx, acq, sc\}$, $mo2 = rlx$*, and there is no acquire action in* left(Cs_2)*;*

then changing

$$Cs_1[r1 = load_{mo1}(x);\ Cs_2[r2 = load_{mo2}(x)]]$$

to

$$Cs_1[r1 = load_{mo1}(x);\ Cs_2[r2 = r1]]$$

is a sound optimisation.

Proof sketch. We will keep the invariant that the event structure is a tree. By Lemma 6, it suffices to show that for all μ,

$$[\![Cs_1[r1 = load_{mo1}(x);\ Cs_2[r2 = load_{mo2}(x)]]]\!]_\mu$$

can transition to

$$[\![Cs_1[r1 = load_{mo1}(x);\ Cs_2[r2 = r1]]]\!]_\mu$$

just through Read after Read merging steps, which preserve the invariant that the event structure is a tree. This can be proved by induction on Cs_2. If the RaR merging steps can be done from each read corresponding to the read to `r2` into a read corresponding to the read to `r1` with given Cs_2 interposed, then they can be done even with a larger Cs_2, as the syntactic restrictions enforce the side conditions on the event structure of the mergings. □

8.5 Constant propagation

As for common subexpression elimination, the aspect of constant propagation dealing with registers is a consequence of Lemma 5. The aspect dealing with memory accesses can be expressed as follows:

Theorem 4. *If* left($\texttt{Cs}_2$) *does not access* `x`*, does not contain any SC access, and*

- *either* $\texttt{mo1} = \texttt{mo2} = \texttt{na}$ *and there is no release action followed by an acquire action in* left($\texttt{Cs}_2$)*;*
- *or* $\texttt{mo1} \in \{\texttt{rlx}, \texttt{acq}, \texttt{sc}\}$, $\texttt{mo2} = \texttt{rlx}$*, and there is no acquire action in* left($\texttt{Cs}_2$)*;*

and `v` *is a constant, then changing*

$$\texttt{Cs}_1\texttt{[store}_{\texttt{mo1}}\texttt{(x,v); Cs}_2\texttt{[r2 = load}_{\texttt{mo2}}\texttt{(x)]]}$$

to

$$\texttt{Cs}_1\texttt{[store}_{\texttt{mo1}}\texttt{(x,v); Cs}_2\texttt{[r2 = v]]}$$

is a sound optimisation.

Proof sketch. Similar, with Read after Write merging steps. □

8.6 Dead code elimination (restricted)

As explained in Section 8.1, removing atomic writes is unsound. Moreover, because we use final values as observations, even removing nonatomic writes is not sound in general, as it can change the final values of locations. This means that only a restricted version of dead code elimination is sound in our setting. The aspects relating to registers are, again, sound by Lemma 5. Reads whose value is ignored can be removed (irrelevant read elimination, in the terminology of Ševčík):

Theorem 5. *If* `Cs` *and* `ss` *do not access* `r1`*, and* $\texttt{mo} \in \{\texttt{na}, \texttt{rlx}\}$*, then changing* $\texttt{Cs[r1 = load}_{\texttt{mo}}\texttt{(x); ss]}$ *to* `Cs[ss]` *is a sound optimisation.*

Proof sketch. Unless the irrelevant read triggers a race in the original program, in which case the optimisation is vacuously sound, it has no effect on the storage subsystem. Because it is irrelevant and not synchronising, it has no effect, apart from itself, on the issuing of memory actions to the storage subsystem by the thread subsystem either. □

Moreover, if a write is guaranteed to be overwritten, then it can be removed. This is the case, for example, if there is syntactically no branching in between two writes:

Definition 80. A context `C` (respectively `Cs`) *contains branching on the path to the hole* when it is of the form `C'[if (e1 cmp e2) Cs else ss]` or `C'[if (e1 cmp e2) ss else Cs]` (respectively `Cs'[...]`).

Theorem 6. *If* $\texttt{Cs}_2$ *does not contain any branching on the path to the hole,* left($\texttt{Cs}_2$) *does not access* `x`*, does not contain any SC access,*

- *either* $\texttt{mo1} = \texttt{mo2} = \texttt{na}$ *and there is no release action followed by an acquire action in* $\text{left}(\texttt{Cs}_2)$*;*
- *or* $\texttt{mo1} = \texttt{rlx}$, $\texttt{mo2} \in \{\texttt{rlx}, \texttt{rel}, \texttt{sc}\}$, *and there is no release action in* $\text{left}(\texttt{Cs}_2)$*;*

then changing

$$\texttt{Cs}_1\texttt{[store}_{\texttt{mo1}}\texttt{(x,e); Cs}_2\texttt{[store}_{\texttt{mo2}}\texttt{(x,e')]]}$$

to

$$\texttt{Cs}_1\texttt{[Cs}_2\texttt{[store}_{\texttt{mo2}}\texttt{(x,e')]]}$$

is a sound optimisation.

Proof sketch. Similar to Read after Read, with Overwritten Write Elimination steps. □

8.7 Reordering

Reordering can result either from instruction scheduling, or as a side effect of larger syntactic optimisations. Once again, Lemma 5 accounts for reordering of register operations. For memory accesses, *re*ordering in syntactic optimisations is accounted for by *de*ordering in the memory model, and can be expressed as follows for writes:

Theorem 7. *If* $\texttt{x}$ *and* $\texttt{y}$ *are two distinct locations, and* $\{\texttt{mo}_x, \texttt{mo}_y\} \subseteq \{\texttt{rlx}, \texttt{na}\}$*, then changing*

$$\texttt{Cs[store}_{\texttt{mo}_x}\texttt{(x,e}_x\texttt{); store}_{\texttt{mo}_y}\texttt{(y,e}_y\texttt{)]}$$

to

$$\texttt{Cs[store}_{\texttt{mo}_y}\texttt{(y,e}_y\texttt{); store}_{\texttt{mo}_x}\texttt{(x,e}_x\texttt{)]}$$

is a sound optimisation.

Proof sketch. While the event structures of the two threads are different, the two threads are weakly bisimilar, where visible actions are those issuing memory actions to the storage subsystem. In particular, in both cases, the later event can be deordered with respect to the earlier event, to lead to the same event structure. Given the memory orders of the writes, they do not obstruct merging, and the conditions on the shape of the event structure of the merging steps do not constrain the order of the writes, so merging steps are not affected. □

The same holds for reordering a read with respect to a write, or a write with respect to a read or a write, for the same reason. For reordering a read with respect to a read, up to one instance of the second read per value the first read can read (see the event structure of the second thread of MP in Section 6.6, but with a non-acquire first read) can be executed; however, only the one corresponding to the value read by the first read matters in the execution.

8.8 Irrelevant read introduction

The memory model also accounts for introducing reads in contexts restricted enough for them to be irrelevant to the execution of the program (irrelevant read introduction, in the terminology of Ševčík):

Theorem 8. *If* $\texttt{Cs}_2$ *does not contain any branching on the path to the hole, there is no acquire action, no SC access, and no access to* `x` *in* $\text{left}(\texttt{Cs}_2)$, `r1` *occurs neither in* $\texttt{Cs}_1$ *nor in* $\texttt{Cs}_2$, *and* `mo` $\in \{\texttt{na}, \texttt{rlx}\}$ *then changing*

$$\texttt{Cs}_1\texttt{[Cs}_2\texttt{[r2 = load}_{\texttt{mo}}\texttt{(x)]]}$$

to

$$\texttt{Cs}_1\texttt{[r1 = load}_{\texttt{mo}}\texttt{(x); Cs}_2\texttt{[r2 = load}_{\texttt{mo}}\texttt{(x)]]}$$

is a sound optimisation.

Proof sketch. Because there is no branching, the second read appears in the event structure if and only if the first does too. Because there is no acquire action, no SC access, and no access to its location, the second write can be deordered up to the first, and executed with it. Therefore, adding the first read does not add any race, and any value the first read can read, the second read can read too. □

8.9 Whole-program optimisations

Having global information about a program enables more aggressive optimisations, as illustrated by the Java Causality Test 1 of Pugh [68] (see Page 78): Pugh justifies this outcome:

> "interthread compiler analysis could determine that `x` and `y` are always non-negative, allowing simplification of `r1 >= 0` to true, and allowing write `y = 1` to be moved early."

We can show that this kind of optimisation based on global value-range analysis is sound in our memory model. For example, we can define a very coarse over-approximation of the values that can arise in a program without arithmetic:

Definition 81. A program is *arithmetic-free* when all expressions are of the form `v` or `r` (but not `e + e'`).

Definition 82. A value is *ground* when it is syntactically present in the source program, or the initial value.

Theorem 9. *If the whole program does not contain any arithmetic, for all value* `v2`*, and for all ground value* `v1`, `v1` $[\![\texttt{cmp}]\!]$ `v2`*, then changing* `C[if (e1 cmp v2) ss1 else ss2]` *to* `C[ss1]` *is a sound optimisation.*

Proof sketch. The constraint ensures that the `else` branch only occurs below reads that can be marked as dead by value-range speculation steps. The proof is along the lines of that of Theorem 10. □

This works as a sanity check that the memory model abstracts enough over conditionals. While it might seem self-evident, it is not true of hardware memory models (see Section 2.5), and the attempt of C/C++11 at abstracting over conditionals is part of the reason it exhibits out-of-thin-air (as discussed Section 2.3.2).

8.10 Testing: per-execution translation validation

The ultimate point of our relaxed semantics is to accommodate syntactic compiler optimisations: given a source program, it will be compiled using a range of syntactic compiler optimisations, and the result executed in a relaxed hardware model; any behaviour of that should be admitted by our source semantics.

To demonstrate concretely and constructively that the steps of our semantics capture the behaviour that syntactic compiler optimisations introduce, we implemented three standard syntactic optimisations for our language: the common subexpression elimination, constant propagation, and dead code elimination passes of Appel [9] (modulo the caveats in Section 8.6). We then instrumented these optimisation passes to record their effects, as captured by the semantics in terms of merging, etc., as annotations in the optimised code. For example, this compiler can perform the following standard common subexpression elimination, eliminating the second read of `y`. In so doing, it records the fact that this is modelled in our semantics by Read-after-Read (RaR) mergings of the events of the loads (the details of this annotation are elided; they involve the events of the event structure of the original program thread).

```
r1 = load_rlx(x);          r1 = load_rlx(x);
if (r1 == 0)               if (r1 == 0)
  r2 = load_rlx(y)    ⇝      r2 = load_rlx(y) RaR ...
else                       else
  r2 = load_rlx(y)           r2 = load_rlx(y) RaR ...
r3 = load_rlx(y)           r3 = r2 // optimised
```

We then developed an instrumented in-order thread semantics for such instrumented code that, when executed, produces an execution of the original thread in our semantics that has the same result. This thus justifies the correctness of the optimisation for the particular execution in question. Continuing the example, running it on the trace "$\mathsf{R}_{\mathsf{rlx}}\,\mathsf{x}\,1$; $\mathsf{R}_{\mathsf{rlx}}\,\mathsf{y}\,1$" produces the trace (eliding details again) "$\mathsf{R}_{\mathsf{rlx}}\,\mathsf{x}\,1$; RaR merging ...; $\mathsf{R}_{\mathsf{rlx}}\,\mathsf{y}\,1$".

We tested this optimising compiler on a series of small hand-written tests, including some manually translated from examples of C program optimisations.

8.11 Related work

Ševčík [86, 74] investigates the soundness of optimisations in DRF-SC (this is discussed further in Section 11.2). Ševčík et al. [76] prove the correctness of a compiler for a C-like language with TSO semantics to x86-TSO, and show the soundness of common compiler optimisations in that setting, though only in forms that are sound thread-locally. Manson et al. [49] show that some optimisations, including reorderings, are sound in the Java Memory Model. However, Cenciarelli et al. [26] and Ševčík and Aspinall [75] show that common subexpression elimination is not sound in Java, but is still performed by Java compilers. Vafeiadis et al. [83] show that many *source-to-source* optimisations are not valid in C/C++11, making it unsuitable to express phases of a compiler. Alglave et al. [5] show that a class of non-relational analyses are sound in relaxed memory models respecting some constraints. Vafeiadis and Zappa Nardelli [85] define two algorithms for removing redundant fences on x86, and prove their correctness. Morisset and Zappa Nardelli [59] define an algorithm for removing redundant fences on x86, ARM, and Power, integrate

it in the LLVM backend, and prove its correctness. Many authors show soundness of optimisations in their own memory model (see Chapter 11).

Morisset and Zappa Nardelli [58] found miscompilations (according to for C/C++11) of concurrent programs by production compilers by comparing sequential memory access traces in unoptimised and optimised programs.

Chapter 9

Additional validation

In this chapter, we show that programs of three specific forms, for which there is a well-established, expected behaviour, actually have that behaviour in our memory model. The arithmetic-free fragment has the "no out-of-thin-air guarantee" of Ševčík [86], the release/acquire fragment matches the strong release/acquire of Lahav et al. [43], and SC fences restore sequential consistency. We also discuss the relation of our memory model with C/C++11.

9.1 "No out-of-thin-air guarantee" for arithmetic-free programs

Syntactically, an arithmetic-free program does not produce new (non-ground) values: they are either hard-coded (ground), or passed around. An elementary sanity check, the "no out-of-thin-air guarantee" of Ševčík [86], is that such a program does not produce new values semantically:

Theorem 10 (no out-of-thin-air guarantee)**.** *In arithmetic-free programs, all executed reads read ground values (as per Definitions 81 and 82).*

We specify "executed", as a read of a non-ground value can be merged into a read or write of a non-ground value; however, neither will be part of the execution proper. In the initial event structures, non-ground values appear in a very restricted way:

Lemma 7. *In the initial event structure of a thread of an arithmetic-free program, all writes of non-ground values are after a read of the same value.*

Proof. By induction on the syntax. □

The threat is that excessive deordering, enabled by excessive speculation steps, would allow a write to somehow escape. We show that "not too much" of the event structure can be marked as dead by exhibiting a live path of arbitrary length to the first non-ground executed write.

Definition 83. A set of events is *live* when there is an execution in which all the events form part of events that are executed (as per Definition 47).

Lemma 8. *In an arithmetic-free program, for every n, there is a live, ground path of length n to one of the events that forms the earliest non-ground executed write in the initial event structure of that write's thread.*

Proof. By induction:

- For the base case, take the empty list. It is of length 0, is ground, live, and one of the events that forms the write is after it (in fact, all are).
- For the induction step, assume we have a list of length n, of ground values, live, and one of the events that forms the write is after it. Therefore, from Lemma 7, as the path is ground, there has to be an event in between the path and the write.
  - If the event is not a read, whether it is merged or executed in the execution where the non-ground write is executed, add it to the list. Because the write is executable, this event is executable too; in particular, if it is a lock or an RMW, then it can be executed: otherwise, the write could not. The new list is of length $n + 1$, ground (because the path is ground, and Lemma 7), live, and the event that forms the write is after it.
  - If the event is a read:
    * if it is merged in the execution where the non-ground write is executed, then as the path is ground, it is ground too. Add this read event to the list. The new list is of length $n + 1$, ground, and live.
    * if it is not, then, as the path is live, one of the branches of the read can be executed. Moreover, it has to be of a ground value. Therefore, at least one of the ground reads cannot be dead. Therefore, by the condition of the deordering rule, there has to be an event that forms the write after that read event. Add this read event to the list. The new list is of length $n + 1$, ground (we have just shown it), live, and the event that forms the write that is after is after it (it might be a different one from the one from the induction hypothesis, though). □

We can now prove Theorem 10:

Proof. By contradiction. Assume a (non-speculated) read reads a non-ground value. The storage subsystem ensures that that value must have been written, so a write of that value must have been executed. Consider the first write of a non-ground value to be executed. By Lemma 8, the write cannot be executed in a finite number of steps. □

9.2 Strong release/acquire

The fragment of C/C++11 restricted to release writes, acquire reads, `sc_fence`s, and RMWs, is simple (Lahav et al. [43] give it a simplified, operational description), usable (it can be used to implement locks; strong program logics have been developed for it [82]), and relatively cheaply implementable (using `lwsync`/`isync` pairs on Power).

Lahav et al. [43] show that it is possible to strengthen the C/C++11 memory model to make it "tight" for release/acquire: all behaviours of a release/acquire program in their strengthened C/C++11 memory model, SRA, are behaviours of its compilation according to the standard compilation scheme to Power (as described in Section 7.2) in the axiomatic Power model of Alglave et al. [7]. Lahav et al. take advantage of this result to propose an equivalent, but simpler operational model for the release/acquire fragment. Instead of doing an indirect proof through the axiomatic model of Alglave et al., we can directly show "tightness" of our model with the reference Power model of Sarkar et al.

Theorem 11 (strong release/acquire)**.** *Let P be a program the memory accesses of which are only release writes, acquire reads, RMWs, and* `sc_fence`*s. Then all the behaviours of P (in our semantics) are behaviours of its compilation to Power according to the standard compilation scheme.*

Proof sketch. Because all the memory actions are release writes, acquire reads, RMWs, and `sc_fence`s, merging is forbidden, and deordering is forbidden except for deordering an acquire read with respect to a release write. However, this can be simulated by delaying the propagation of the release write, that is, the `lwsync` and the write, in the storage subsystem. □

This makes the GPS program logic of Turon et al. [82] sound for the release/acquire fragment of our memory model.

9.3 Fences

Unlike in C/C++11 [83], adding `sc_fence`s in between every two memory accesses does restore sequential consistency:

Theorem 12. *Let P be a program such that all its memory accesses are atomic, and there is an* `sc_fence` *in between any two accesses in any path of the program. Then all of the behaviours of P are sequentially consistent.*

Proof. Because all accesses are atomic, there are no data races. The property then follows from the treatment of `sc_fence`s by the storage subsystem [13], as they prevent any deordering and merging in the thread subsystem. □

9.4 Relation to C/C++11 (for programs with only relaxed accesses)

While, as C/C++11 is the state of the art in established memory models, it would be interesting to show that our memory model is sound with respect to C/C++11, several aspects of C/C++11 hinder this:

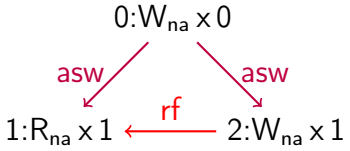
- the strange handling of races (explained below, in Section 9.4.1);
- the fact that that the memory model of C/C++11 cannot handle thread-local sources of undefined behaviour [12], as described in Section 1.2.4;
- the fact that the behaviour of SC accesses in C/C++11 is more restricted than the behaviour of their expected compilation on Power and ARM [44, 47].

9.4.1 Race reconstruction

In C/C++11, surprisingly, the candidate execution corresponding to a racy execution is not necessarily a consistent candidate execution. For example, the program below contains a race between the read of `x` by the first thread, and the write to `x` by the second thread:

```
          x = 0
-------------------------
r1 = load_na(x) || store_na(x,1)
```

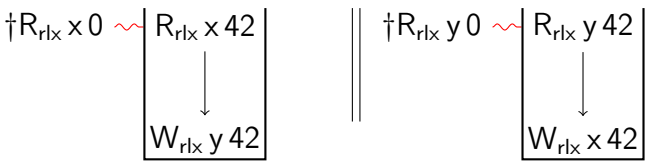
However, the execution witnessing this race in the C/C++11 memory model is the one where the read reads 0 from the initial state. The execution in which the race actually happens, and for which our model flags the race, is the one where the read reads 1. However, this execution is not consistent according to C/C++11, because the read and the write are not related by the C/C++11 happens-before relation.



This is arguably a design flaw of C/C++11 [83, 65]: the presence of races is determined on consistent candidate executions, but the consistency predicate filters out the actual execution exhibiting the race.

9.4.2 Value-range speculation in C/C++11

By considering executions individually, C/C++11 implicitly assumes that any value-range speculation is valid. Considering an individual pre-execution corresponds, in our memory model, to speculating that any execution other than the current one is impossible, without checking, and only looking whether it is actually possible to read the predicted values. For example, the configuration that corresponds to the pre-execution of LB+ctrldata+ctrl-single that exhibits the thin-air behaviour would correspond to an execution of our memory model where reading values other than 42 has (incorrectly) been speculated to be impossible:



9.4.3 Relating executions

Because of the issues above, we restrict our attention to programs where accesses are all relaxed, and without thread-local sources of undefined behaviour.

Definition 84. A program is a *relaxed-only* program when its memory actions are only relaxed accesses, and it is not of the form `C[undef]`.

We can now show how to reconstruct a C/C++11 consistent execution from an execution of a relaxed-only program in our memory model. We proceed in two phases: first, we reconstruct a pre-execution, and then an execution witness.

Pre-execution The whole-program induced configuration described in Section 4.3 corresponds to a pre-execution, up to the initial writes, which are implicit in our memory model.

Definition 85. The *restriction* of an event structure $\langle E, \leq, \sim, \lambda \rangle$ to a subset E' of E is $\langle E', \leq \cap (E' \times E'), \sim \cap (E' \times E'), \{(e, a) \in \lambda \mid e \in E'\} \rangle$.

For a configuration, $\sim$ is always empty, so we can define:

Definition 86. The *restriction* of an event structure $\langle E, \leq, \sim, \lambda \rangle$ *to one of its configurations* C is $\langle C, \leq \cap (C \times C), \{(e, a) \in \lambda \mid e \in C\} \rangle$.

Definition 87. The *sb-projection* of

- a per-thread pre-execution $\langle E, sb, dd, \lambda \rangle$ is $\langle E, sb, \lambda \rangle$;
- a whole-program pre-execution $\langle E, sb, dd, asw, \lambda \rangle$ is $\langle E, sb, \lambda \rangle$;
- a candidate execution is that of its pre-execution.

Lemma 9. *The restriction of the whole-program event structure of a program without* `undef` *to any of its (whole-program) induced configurations is (up to graph isomorphism) the sb-projection of a C/C++11 pre-execution of that program, without the initial writes.*

Proof. By induction on the statements `ss` of the threads, for any μ, μ such that $\mathrm{dom}(\mu') = \mathrm{dom}(\mu)$ and $\forall (r, v) \in \mu.\, \exists N.\, (r, (v, N)) \in \mu'$, any configuration of $[\![\texttt{ss}]\!]_\mu$ is the sb-projection of ${}^{\mathrm{C}}[\![\texttt{ss}]\!]^{\mathrm{t}}_{\mu'}$. □

Definition 88. The *induced pre-execution* of a whole-program configuration C of a program without `undef` P is the unique (up to graph isomorphism) pre-execution X of P such that its sb-projection, without the initial writes is C.

Execution witness

Definition 89. A trace t *induces a partial execution witness* W by induction on the trace:

- The empty trace induces the empty execution witness.
- When receiving a read request, this induces an 'rf' edge from the write that satisfies it to the read being satisfied. The write can be an initial write.
- When a write reaches coherence, this induces an 'mo' edge, as per *coherence*.
- When executing a lock (a load-linked/store-conditional pair), this induces an 'lo' edge from the unlock (that the load-linked read from: the unlock is a write in the storage subsystem) to the lock (which is a store-conditional in the storage subsystem).

The functioning of the Power storage subsystem then enforce invariants about partial execution witnesses:

Lemma 10. *For any partial execution witness induced by a trace of a relaxed-only program:*

- *If* $(a, b) \in rf$*, then* a *and* b *are at the same location.*
- *If* $(a, b) \in mo$*, then* a *and* b *are at the same location.*
- *If* $(a, b) \in mo$*, then* b *is not an initial write.*

- *If* $(a, b) \in hb$*, and* a *is not an initial write, then* $(a, b) \in sb^+$*.*
- *If* $(a, b) \in hb$*, and* a *and* b *are at the same location, then* a *is executed before* b*.*

Proof sketch. For a relaxed-only program, hb is $(sw \cup sb)^+$ which is $(asw \cup sb)^+$, where asw is the spawning of the threads. Lemma 2 can then be used to transfer the invariants of the Power storage subsystem [72]. □

Definition 90. The *restriction* of a pre-execution $\langle E, sb, dd, asw, \lambda \rangle$ to a subset E' of E is $\langle E', sb \cap (E' \times E'), dd \cap (E' \times E'), asw \cap (E' \times E'), \{(e, a) \in \lambda \mid e \in E'\} \rangle$.

Definition 91. The set of events of a partial execution witness $\langle rf, mo, lo, sc \rangle$ is

$$\{e \mid \exists e'. \{(e, e'), (e', e)\} \cap (rf \cup mo \cup lo \cup sc) \neq \varnothing\}.$$

Definition 92. The *restriction* of a pre-execution X to a (partial) execution witness W is the restriction of X to the set of events of W.

Lemma 11. *The partial execution witness of a trace of a relaxed-only program, together with the restriction of the induced pre-execution to that partial execution witness, is a consistent execution.*

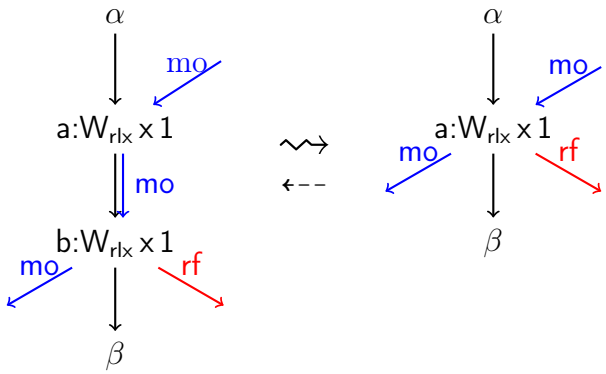
Proof sketch. We consider the conjuncts of the C/C++11 consistency predicate:

- *assumptions*: all relations have finite prefixes, as everything is finite; moreover, *asw* is on different threads, as it is only from the initial writes (from thread 0) to actual threads (as per Definition 16).
- *well_formed_threads*: C/C++11 bookkeeping.
- *well_formed_rf*: *rf* is a relation on the actions of the pre-execution at the same location, where the source is a write and the target a read, and values match, and the symmetric of rf is functional.
- *locks_only_consistent_locks*, *locks_only_consistent_lo*: bookkeeping for locks, which we do not consider here.
- *consistent_mo*: *mo* is a transitive, irreflexive relation on the actions of the pre-execution such that two actions are related when they are different writes at the same atomic location.
- *sc_accesses_consistent_sc*, *sc_fenced_sc_fences_heeded*: for SC accesses and fences, which we do not consider here.
- *consistent_hb*: the transitive closure of *hb* is irreflexive: here, *hb* is $(sw \cup sb)^+$, which is $(asw \cup sb)^+$, which is irreflexive by construction (in Definition 16).
- *consistent_rf*:
  - *det_read*: a read has a read-from edge only if there is a write at the same location that happens-before it: here, the initialisation write.
  - *consistent_non_atomic_rf* : concerns NA accesses, which we do not consider here.

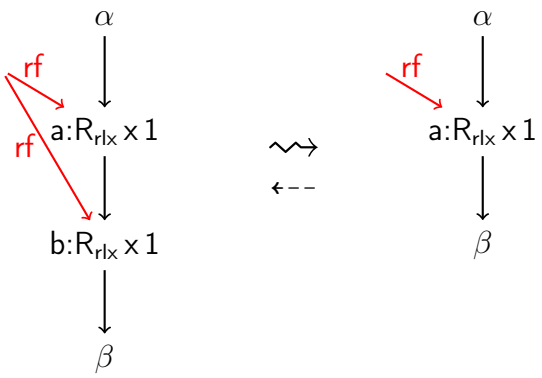
- *consistent_atomic_rf* : Assume $(w, r) \in rf$, r is atomic, and $(r, w) \in hb$.
  From $(w, r) \in rf$, w and r are at the same location.
  From $(w, r) \in rf$, r is not an initial write. With $(r, w) \in hb$, by Lemma 10, we have $(r, w) \in sb^+$.
  With w and r being at the same location, by Lemma 2, we have that r is executed before w.
  Therefore, r cannot read from w, given that when r is executed, w has not yet been executed.
- *coherent_memory_use*: By Lemma 10, a violation would imply a violation of the corresponding coherence condition for the storage subsystem.
- *rmw_atomicity*: concerns RMWs, which we do not consider here.
- *sc_accesses_sc_reads_restricted*: concerns SC accesses, which we do not consider here.

- *errors*:
  - *unsequenced_races*: concerns non-atomic accesses, which we do not consider here.
  - *data_races*: concerns non-atomic accesses, which we do not consider here.
  - *indeterminate_reads*: all reads have an *rf* edge.
  - *locks_only_bad_mutexes*: concerns locks, which we do not consider here. □

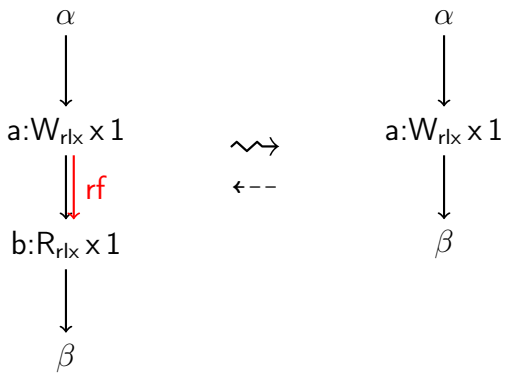
This execution witness is partial, because it only concerns memory actions that are issued to the storage subsystem. Merged memory actions have to be inserted back into the execution witness. For example, if a Write after Write merging turns ($\rightsquigarrow$) the event structure fragment (in black) on the left to the one on the right, then the partial execution witness (in colour) on the right can be completed ($\dashrightarrow$) into the one on the left:



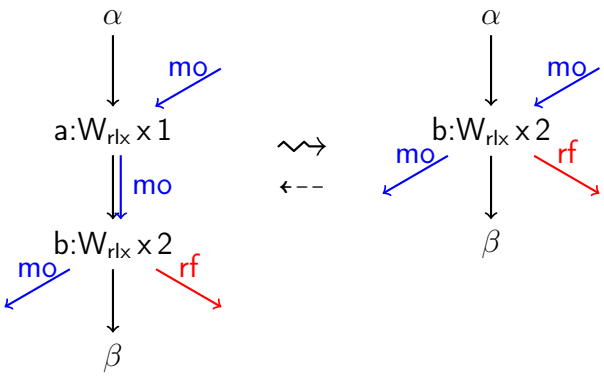
For Read after Read mergings, the merged inherits the reads-from of the read it is merged into (With non-atomics, we could have an interposed WaR which would complicate the issue):



For Read after Write, the read reads-from the write:



For Overwritten Write Elimination:



We express the completion in terms of mo_1 rather than mo (see Definition 19):

Definition 93. The *completion* of a partial execution witness $\langle rf, mo_1, \varnothing, \varnothing \rangle$ with respect to a merging is defined by:

- if e_1 is RaR-merged into e_0:

$$\langle rf \cup \{(w, e_1) \mid (w, e_0) \in rf\}, mo_1, \varnothing, \varnothing \rangle$$

- if e_1 is RaW-merged into e_0:

$$\langle rf \cup \{(e_0, e_1)\}, mo_1, \varnothing, \varnothing \rangle$$

- if e_1 is WaW-merged into e_0:

$$\left\langle \begin{array}{l} (rf \setminus \{(e_0, r) \in rf\}) \cup \{(e_1, r) \mid (e_0, r) \in rf\}, \\ (mo_1 \setminus \{(e_0, w) \in mo_1\}) \cup \{(e_0, e_1)\} \cup \{(e_1, w) \mid (e_0, w) \in mo_1\}, \\ \varnothing, \\ \varnothing \end{array} \right\rangle$$

- if e_0 is OWE-merged into e_1:

$$\left\langle \begin{array}{l} rf, \\ (mo_1 \setminus \{(w, e_1) \in mo_1\}) \cup \{(e_0, e_1)\} \cup \{(w, e_0) \mid (w, e_1) \in mo_1\}, \\ \varnothing, \\ \varnothing \end{array} \right\rangle$$

- There is no case for WaR, because we do not consider non-atomic accesses.

The *lo* and *sc* components are empty because we do not consider locks and SC accesses.

We can then show that completion preserves consistency; however, the statement has to take into account the fact that from the point of view of C/C++11, the original and completed partial execution witnesses do not necessarily correspond to pre-executions of the same program, and neither has to be a pre-execution of the program for which we are constructing an execution witness:

Lemma 12. *Given a set of events E, the completion of a partial execution witness with set of events E with respect to a merging into events of E of a relaxed-only program which is part of a consistent execution is part of a consistent execution.*

Proof sketch. As for Lemma 11, we inspect the consistency predicate. The completions all insert a memory action in *sb*, and therefore in *hb*. □

Theorem 13. *Any (whole-program) induced configuration of a relaxed-only program is the sb-projection of a C/C++11 consistent execution of that program (up to initial writes).*

Proof sketch. Consider a trace of our memory model. The trace induces a pre-execution, and a partial execution witness which form a consistent execution (but not necessarily of that program). We then complete this partial execution witness with the mergings into events of the partial execution witness, as per Definition 93, starting from the end of the trace. By starting from the end of the end of the trace, we insert back the events that are part of the execution. (A merging not into events of the partial execution witness would introduce events that are not part of the execution. Conversely, the non-last merging into events of the partial execution witness may refer to events that do not appear in the partial execution yet, but would if more mergings were completed. For example if c is merged into b, which is merged into a, completing the execution containing just a with respect to the merging of c into b will refer to b despite it not appearing in the partial execution witness, and we would not be able to determine whether b belongs in the execution or not.) The pair of that pre-execution and the completed partial execution witness (which is not partial anymore) is a consistent execution of the program. □

9.5 Relation to strengthened C/C++11

We conjecture that the sb-projection of any consistent executions of the C/C+11 memory model where consistency also requires that $po \cup rf$ is acyclic is an induced configuration in our memory model.

Chapter 10

Limitations

While our memory model tackles many issues concerning the design of a memory model for a programming language, we have still made some simplifying assumptions that would have to be lifted to tackle a full-fledged programming language. In this chapter, we discuss how these limitations could be lifted.

10.1 Finite value domain

Our memory model assumes that the domains of the variables are finite. This is not a limitation for a programming language, but could be for reasoning. In fact, in programming languages like C/C++11, data types are finite.

However, it is sometimes useful for reasoning to assume for example that `int` $= \mathbb{Z}$. In our memory model, this will make some steps inaccessible, requiring for example an infinite number of mergings before a deordering can be performed. One possible way to go around this limitation would be to allow (a finite number of) infinite families of non-overlapping steps. This makes it possible to do, for example, a merging in an infinite number of branches, without making memory actions "at infinity" (after a non-terminating loop) accessible, which would be wrong.

10.2 Loops

Our memory model assumes there are no loops in the program. We believe that there is no theoretical obstacle, as loops can simply be unfolded:

$$[\![\texttt{while e1 do e2; ss}]\!]_\mu = [\![\texttt{(if e1 then (e2; while e1 do e2)); ss}]\!]_\mu$$

However, taken literally, this definition would generate an event structure with an infinite set of events. To keep the model executable, we would need to do this unfolding dynamically, on a per-need basis, which would obscure the semantics. We therefore did not include loops in our calculus to keep the Lem formalisation straightforwardly executable.

10.3 Addresses and address dependencies

The language we consider enforces that all addresses are fixed upfront in the program, and are not computed. This makes it impossible to express address dependencies. This

was done to ensure straightforward executability of the memory model. However, this restriction can be lifted by resolving addresses in the same way we resolve "normal" data. This would match C/C++11, where there is only one relation, 'dd', that encompasses both address and data dependencies.

10.4 Location typing

For simplicity, we assume a location typing: any location is used either for atomic accesses, for nonatomic accesses, or for lock accesses. The formalised C/C++ memory model [15] makes the same assumption.

However, this assumption is too restrictive in practice. It is a common idiom for a thread to access a piece of data atomically in a portion of a program where it is shared, and then nonatomically once it has ensured it is the only thread accessing the data, as described for example by McKenney [52, Quick Quiz 4.15 answer, p. 356]. Moreover, to be able to implement locks in the language, lock locations have to be accessible using reads and writes.

To lift this restriction, the memory model would likely need to be extended with rules about merging nonatomic and atomic accesses, which makes the detection of races more complicated.

10.5 Read-modify-writes

Our read-modify-writes correspond to C/C++11 RMWs with an acquire/release success memory order, and acquire failure memory order. This restriction is to keep the model manageable. We do not know of any problems with other kinds, but have not worked out the details.

10.6 Locks and divergence

Locks are implemented on hardware using loops around RMWs (which are themselves implemented by load-link/store-conditional pairs on some RISC architectures), which can spuriously fail (see Section 7), and can therefore diverge (in fact, hardware typically does not guarantee progress). However, to keep our formalisation straightforwardly executable, locks feature an explicit failure state (like C/C++11) that somehow corresponds to the lock guessing it will never succeed. An alternative would be to allow threads to get stuck at locks.

10.7 Memory layout model

Our calculus features a single data type. Extending it with compound data types raises the question of the layout of the memory [57]. Moreover, it opens up the possibility of "mixed-size accesses": accessing overlapping locations using accesses of different widths, which interacts with concurrency, see Flur et al. [33].

10.8 Unsequenced and indeterminately sequenced accesses

In our calculus, program order imposes a total order on the memory actions of a thread. This order can then be weakened by deordering steps. However, in C/C++11, it is possible to have *unsequenced* memory actions. For example, in

```
x + x
```

there is no order between the two NA reads. Moreover, it is also possible to have non-determinism. For example, if we wrap a load of `x` in a function `f`, there in a non-deterministic order between the two calls of `f`, and thus between the two NA reads;

```
int f(void) {
  return x;
}

f() + f()
```

While the latter could be dealt with by enumerating all the orders, and considering their executions, the formalisation relies, for simplicity, on the fact that program order is a total order.

These features would interact strangely with value-range speculation.

10.9 Sequencing of threads

A design choice in a memory model is whether it should be stable under sequencing of threads (also called "linearisation"), that is, replacing two threads in the source program by their concatenation. On the one hand, it is a reasonable thing for a programmer, and possibly a compiler, to do, but on the other, it has surprising consequences. Vafeiadis et al. [83] show that sequencing is unsound as a source-to-source transformation in C/C++11, and propose changes to C/C++11 in which it is sound. The Causality Test Cases 19 and 20 of Pugh [68], discussed in Section 6.3, feature a `join` operator that has the effect of sequencing threads, but not of synchronisation, so that writes from the later thread can be executed before the earlier thread finishes executing. In our setting, sequencing of threads is unsound, as illustrated by example Causality Test Case 19 (without the `join` operator). However, if sequencing is deemed desirable, we could introduce a step matching the effect of the `join` operator. The forbidden behaviour of Causality Test Case 5 discussed in Section 6.3 would also become allowed.

10.10 Optimising synchronising accesses

As explained in Section 8.1, value-range analysis makes seemingly benign optimisations on synchronising accesses unsound. There is a tension between how much we want to allow optimisations to take advantage of value-range analysis, and how much we want compilers to be allowed to optimise synchronising accesses.

Bastien notes that although he is unaware of any compiler performing the optimisation, it is being considered by compiler writers [10]. If compilers move towards a model where

synchronising accesses are being optimised, one possible way to address it would be to include merging rules for synchronising accesses, leaving fences where the access used to be in the event structure. The situation is likely to be more intricate than for non-synchronising accesses. Compromises might have to be made between which optimisations are allowed, how much optimisation is allowed, and the simplicity of the model. However, in the absence of ground truth to guide these choices, designing rules for this would be challenging, and best done with input from compiler writers.

10.11 Monotonicity

A memory model is monotone (Vafeiadis et al. [83]) when strengthening memory accesses (according to Definition 40) does not introduce extra behaviour. This is a desirable property, because it allows the programmer to be cautious and over-synchronise the program, while only paying a penalty in execution time. However, because of value-range analysis, our memory model is not monotone, as shown in Section 8.1.

This could be addressed in several ways. A first, straightforward approach would be to add a strengthening step to the semantics that changes the memory order of an event. This would directly restore monotonicity, but does not address the underlying problem of the interaction between merging and value-range speculation. Another approach would be to address this problem by extending the merging rules to allow merging of synchronising accesses, as discussed in the previous section.

10.12 The `consume` memory order

The heart of the RCU (read-copy update) synchronisation mechanism [52] is the preservation by Power and ARM of address dependencies, including "artificial" address dependencies, as illustrated by the Power MP+lwsync+addr litmus test:

x = y = 0	
$\texttt{store}_{\texttt{rlx}}\texttt{(x,1);}$	$\texttt{r1 = load}_{\texttt{rlx}}\texttt{(y);}$
$\texttt{store}_{\texttt{rel}}\texttt{(y,1)}$	$\texttt{r2 = load}_{\texttt{rlx}}\texttt{(x+(r1 xor r1))}$

$\texttt{r1 = 1} \wedge \texttt{r2 = 0}$ forbidden on Power

```
PPC MP+lwsync+addr
{
0:r2=x; 0:r4=y;
1:r2=y; 1:r5=x;
}
 P0            |  P1           ;
 li r1,1       |  lwz r1,0(r2) ;
 stw r1,0(r2)  |  xor r3,r1,r1 ;
 lwsync        |  lwzx r4,r3,r5;
 li r3,1       |               ;
 stw r3,0(r4)  |               ;
```

The actual address of the second read, `x` does not depend on the address of the first read of `y`. However, Power respects the syntactic dependency, and ensures that the read of `x` is executed after the read of `y`. This idiom is used pervasively throughout the Linux kernel,

with more than 10000 instances [53]. To support it at the level of the programming language, the `consume` memory order was introduced in C/C++11. However, because requiring compilers to preserve source dependencies is problematic, `consume` has not been actually implemented as desired yet. Instead, compilers implement it as the stronger `acquire` [81]. Its formalisation in C/C++11 has proved to be unsatisfactory, and is currently being revised [54, 55].

Our approach takes an approach incompatible with the consume memory order and RCU by erasing syntactic dependencies during the construction of the event structure (like compilers). It is not clear how these dependencies could be preserved. One approach would be to label the causality *edges* of the event structure with additional information, and to adapt the rest of the rules.

10.13 The SC memory order

Our treatment of SC accesses as their compilation to Power is very ad-hoc. While it makes the leading-sync compilation scheme (as in Section 7.2, that is, with a `sync` before the memory access) correct by construction, it is unclear whether the trailing-sync compilation scheme (with a `sync` after the memory access) is correct.

10.14 Fences

Release and acquire fences could be supported by our model by adding two new memory actions: a release fence would be treated thread-locally just like a release write, whereas an acquire fence would be treated like (a single event of) an acquire read, with less deordering allowed. Merging rules would have to be adapted.

10.15 ARM

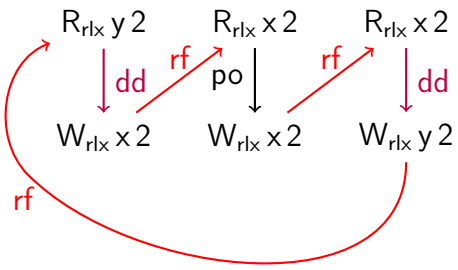
When we started our work, the memory model of ARM had been the subject of less attention than the Power and x86 memory models. While there were a number of litmus tests where ARM did not match with Power, their status was unclear. A number of changes took place during our work.

Non-multiple-copy-atomic ARM Flur et al. [32] developed a new, special-purpose memory model for ARMv7 and v8 (as described in Section 2.5.2). This new memory model allows the following litmus test, ARM-weak (related to the WWC+datas litmus test), to end with `r2 = 2`, despite this behaviour not being currently observable on any ARM hardware (because all implementations are multi-copy-atomic):

`x = y = 0`

`r1 = load`$_{\texttt{rlx}}$`(y);`	`r2 = load`$_{\texttt{rlx}}$`(x);`	`r3 = load`$_{\texttt{rlx}}$`(x);`
`store`$_{\texttt{rlx}}$`(x,r1)`	`store`$_{\texttt{rlx}}$`(x,2)`	`store`$_{\texttt{rlx}}$`(y,r3)`

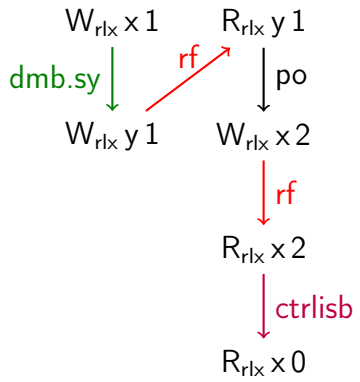
`r2 = 2` ?



Test ARM-weak: Allowed on ARMv7

The justification in their memory model is as follows: with a storage subsystem as in Figure 2.6, Thread 2 can issue the read (but not satisfy it yet, which is possible on ARM, but not on Power, nor with our memory model) and the write, and make them both flow down to the queue it shares with thread 3. Thread 3 can then execute its read of x, and make it flow down the queue to meet the write of 2 to x to be satisfied, then write 3 to y, and flow it all the way down to main memory, past the memory actions of thread 2 on x. Thread 1 can then execute its read of y, and make it flow all the way down to main memory to read 2, then execute its write of 2 to x, and make it flow all the way down to main memory. Thread 2 can then flow its read of 2 to main memory, to read 2.

Multiple-copy-atomic-ARM ARM then retrospectively changed the definition of ARMv8 (as described in Section 2.5.3), which is now required to be multi-copy-atomic, forbidding this outcome. However, the effect issuing reads before determining their value still leads to behaviour that is not allowed by the operational model of Power, for example the MP+dmb.sy+fri-rfi-ctrlisb litmus test below, which is architecturally allowed and observed on implementations (see Pulte et al. [69]):



Test MP+dmb.sy+fri-rfi-ctrlisb: Allowed on ARMv7 and v8, and observed

Supporting ARM If we want to support the non-multiple-copy-atomic version of the ARM architecture and the ARM-weak litmus test, our memory model could maybe be adapted to use a variant of the POP storage subsystem of Flur et al., extended to support a Power-like `lwsync` barrier. To support either version of ARM, there is a mismatch to overcome: our memory model relies on being able to satisfy reads immediately, whereas the ARM storage subsystem model enables reads to flow in the memory, and following writes to execute before the read is satisfied. This could maybe be addressed by splitting reads into two parts: a read issue event that does not fix a value, before a family of read satisfy events in immediate conflict, with one for each value.

Chapter 11

Related work

We discussed the C/C++11, Java, and hardware (x86-TSO, Power, ARM) memory models in Chapter 2, and specific points in their respective chapters. In this chapter, we give an overview of the other related work.

11.1 Out-of-thin-air

The following memory models attempt to address the out-of-thin-air problem.

Outlawing ghosts Boehm and Demsky [21] consider a restriction of the C/C++11 memory model that forbids load-store reordering (see the "load buffering" litmus tests) by requiring $hb \cup rf$ to be acyclic. This forbids the relaxed outcome of LB (see Page 76) which is architecturally allowed on Power and ARM, actually observed on some ARMv7 hardware, and enabled by compiler optimisations that reorder independent reads and writes to different locations. Therefore, forbidding load-store reordering would require introducing memory barriers when implementing relaxed accesses, and limiting compiler optimisations. It is currently unclear whether the cost of this would be prohibitive.

Vafeiadis and Narayan [84] consider this strengthened memory model, and show the soundness of a program logic with a non-trivial rule for relaxed reads.

OHMM The operational happens-before memory model of Zhang and Feng [91] is an operational model that features a *replay* mechanism to account for the propagation of writes at different times to different threads. While this mechanism works well on some examples, it does not relate to operational intuitions of how hardware and compilers operate. Moreover, according to the authors, it does not entirely forbid out-of-thin-air behaviour [31].

Theory of memory models The "theory of memory models" of Saraswat et al. [70] represents a program using a plain graph, where nodes represent symbolic memory actions, the value of which is given by an expression over the input it gets from program-order-predecessor memory actions, and edges represent (weakened, modified) program order. Relaxed behaviour is introduced by steps that transform this graph by merging, splitting, etc. Because of their representation of a program, they can only support a language with very restricted control flow, where conditionals cannot contain arbitrary code. Our approach lifts their restriction by using event structures, where conflict allows expressing programs where there are incompatible memory actions.

Generative operational semantics The "generative" operational semantics of Jagadeesan et al. [40] uses *speculation points* to introduce relaxed behaviour. A speculation point can be inserted at any point in the abstract syntax tree; the speculation point then contains two copies of that part of the AST, the initial and the final, where the latter can assume it has seen a sequence of writes. Other parts of the AST can then, under some conditions, move in and out of the speculated part. In particular, writes that occur in both the initial and the final parts of a speculation point can move out of the speculation point. As they are proposing a replacement memory model for Java, they do not feature the release and acquire memory orders. They argue that some basic optimisations are sound in their memory model. It is unknown whether their memory model can be compiled using the expected compilation schemes.

Operational aspects The memory model of Podkopaev et al. [67] is similar to that of Jagadeesan et al. The memory model of Podkopaev et al. is an operational model in which relaxed behaviour is implemented by equipping each thread with an *operations buffer* that has a nesting structure, and having each thread maintain a *viewfront* of which writes it has seen. The operations buffer accounts for speculative executions, and the viewfront for propagation effects. A write that occurs in both branches of an 'if' in an operations buffer can be *promoted* out of the 'if', as the write occurs in all executions of the 'if'. One of the aims of their semantics is to be efficiently executable. It covers all the C/C++11 access memory orders, but not all C/C++11 fences. It is unknown whether it can be compiled using the expected compilation schemes.

Game model of Java The memory model of Jeffrey and Riely [41] revisits the approach of the Java memory model. An execution is allowed when there is a sequence of partial executions (that is, configurations of the event structure of the program, as in Section 4.3), starting from the empty partial execution, such that any adversarial completion of a partial execution of the sequence justifies the next partial execution of the sequence. Considering all completions of a partial execution, as opposed to the JMM considering a single completion, ties in more closely with intuitions of how the hardware and compiler optimisations operate. They show DRF-SC for their base model, and propose an extension that supports reordering of reads.

Promising semantics The memory model of Kang et al. [42] is an operational model in which threads can *promise* (issue early) writes as long as they can *fulfil* the promise (actually issue the write) by continuing to execute in the current memory, without interacting with other threads. When a thread takes a step, it needs to revalidate all its promises by showing it can fulfil them in the new memory. This constrains the reads it can do. This mechanism is reminiscent of our value-range speculation steps, but relies on writes being always possible, rather than reads never being possible, and on thread-local validity of the promise, rather than whole-program validity of the speculation in our memory model. To ensure coherence, writes are timestamped, each thread keeps a view of the latest timestamp it has seen for each location, and reads can only read from writes at the same location with increasing timestamps. The focus of this memory model is thread-local optimisations, and compatibility with the ARM architecture, so they allow the unobserved but (at the time) architecturally allowed behaviour described in Section 10.15 that we forbid. The fulfilment of promises without interaction with other threads means that many thread-local optimisations are sound by design. They also show

the soundness of some basic reordering optimisations. It covers all the memory orders of C/C++11, including fences, except SC. They show it can be compiled to x86 and Power using the expected compilation schemes. Most of their results are mechanised. Podkopaev et al. [66] show that it can be compiled to ARM using the expected compilation scheme.

11.2 Miscellaneous

The following memory models do not attempt to directly address the out-of-thin-air problem, but address related issues.

Sequential consistency Singh et al. and Marino et al. [78, 51] show that, surprisingly, it is possible to adopt sequential consistency as a memory model, and yet incur only a relatively small performance impact for many common workloads. By choosing to support only SC, while they need to limit the usual hardware and compiler optimisations to remain SC-preserving, they can implement unusual, aggressive optimisations. For example, they restrict the optimisations of the LLVM compiler to be SC-preserving, but extend LLVM with a speculative execution mechanism that, to preserve SC, relies on interference checks enabled by additional instructions provided by their purpose-built SC hardware. However, it is unclear what the effect of this approach would be on high-performance systems code; moreover, mainstream hardware is not sequentially consistent.

Trace model of DRF-SC Ševčík describes a trace-based memory model to show the soundness of compiler optimisations in a DRF-SC setting. His memory model describes the semantics of each thread as a set of memory-access-event traces, closed under primitive semantic changes: reordering, elimination, and introduction of memory actions. Wildcard traces, containing wildcard read events $\mathsf{R}\,x\,*$, are used to express independence of a trace's validity on the value that the wildcard reads might read; this captures some of the intensional structure of each thread. Our event structure semantics captures more of that structure, and one can see our previous work [12, §6] as intermediate between the two.

Theory of speculative computation Petri and Boudol [23] develop an operational semantics with explicit speculation steps, where a thread is allowed to guess the value of a read, to show the soundness of some speculation-based optimisations in a DRF memory model.

Cooking the books Petri et al. [64] analyse the requirements for the implementation of the Java memory model.

LLVM event structure model Chakraborty and Vafeiadis [27] develop a model of a fragment of the concurrency behaviour of LLVM intermediate representation using event structures. They do not cover relaxed accesses.

Plan B Demange et al. [29] propose Plan B, a memory model for Java, but one oriented towards implementation above the relatively simple x86-TSO memory model, without the load-store reordering of the Power and ARM architectures that (when combined with compiler optimisations) makes the thin-air problem challenging.

Hardware Hardware memory models (see Section 2.5) are typically thin-air-free, as hardware generally does not feature observable value speculation. However, they are not sound with respect to common compiler optimisations (collapse of conditionals, CSE, and so on).

Instantaneous instruction execution models Zhang et al. [90] present a series of hardware memory models frameworks based on operational semantics featuring various buffers and timestamps, where the execution of individual instructions is atomic. They are able to account for SC and TSO, as well as some less mainstream memory models, and to approximate Power and ARM. Similarly to well-established hardware memory models, they are not sound with respect to common compiler optimisations.

Denotational models of hardware Castellan [25, 24] proposes denotational models of two hardware memory models: a toy architecture and TSO. For these architectures, each thread can be represented by a maximally deordered event structure. The configurations of the synchronised product of the event structures of the thread with that of the storage subsystem correspond to executions, in keeping with traditional semantics by event structures.

Operational reformulation of C/C++11 Nienhuis et al. [62] present a provably equivalent reformulation of C/C++11 with a more operational flavour. This allows them to integrate their memory model with the thread semantics for C11 of Memarian et al. [57]. To exhibit relaxed C/C++11 behaviour, and in particular to allow out-of-thin-air, their memory model includes symbolic steps, where a read can read a symbolic value, to be determined later, possibly via a cyclic justification.

Chapter 12

Conclusion

We have shown that it is possible to design a memory model for a programming language that is weak enough to be sound with respect to hardware- and compiler-induced behaviour, yet strong enough to be usable, and in particular excludes out-of-thin-air behaviour. We support this thesis with an operational semantics where

- threads can take steps that abstract over hardware- and compiler-induced transformations:
  - deordering a memory access with respect to independent program-order predecessors;
  - merging a memory access into redundant memory accesses;
  - speculating that a certain read cannot read a certain value; and
- threads communicate via a non-multiple-copy-atomic storage subsystem that preserves coherence.

It is ugly, but it is better than the alternative: having to program against the details of the hardware and the compiler.

We show that it meets some design goals:

- from testing on classic out-of-thin-air tests, Java causality tests, etc., it seems extensionally good;
- from inspection, it seems (relatively) intensionally good: it can be reasoned about in the same way a compiler can be reasoned about, because it is built around the same intuitions;
- it can be compiled to x86-TSO using the "strong" compilation scheme, and to Power using the expected compilation scheme;
- a family of common compiler optimisations is sound, including some non-thread-local optimisations; and
- it exhibits the expected behaviour on some fragments where the expected behaviour is clear: the release/acquire fragment behaves like its compilation to Power (strong release/acquire), arithmetic-free programs do not exhibit out-of-thin-air behaviour, SC fences restore sequential consistency, and all relaxed-only programs behave like C/C++11 relaxed-only programs (but not the other way around!).

The main disadvantage is that because of the lack of monotonicity and of the value-range speculation steps, it is difficult to reason about.

Index of litmus tests

Bibliography

[1] `concurrency-interest` mailing list. `http://cs.oswego.edu/pipermail/concurrency-interest/`.

[2] *Programming Languages — C.* 2011. ISO/IEC 9899:2011. `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[3] S. V. Adve and M. D. Hill. Weak ordering - A new definition. In *ISCA*, 1990.

[4] V. S. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *ISM*, 2003.

[5] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In *APLAS*, 2011.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[7] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *PLDI*, 2014.

[8] F. E. Allen. Program optimization. In Mark I. Halpern and Christopher J. Shaw, editors, *Annual Review in Automatic Programming*, volume 5. Pergamon Press, New York, 1969.

[9] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[10] J. F. Bastien. N4455 No sane compiler would optimize atomics, April 2015. available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4455.html`.

[11] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[12] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *ESOP*, 2015.

[13] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[14] M. Batty, S. Owens, J. Pichon-Pharabod, S. Sarkar, and P. Sewell. CppMem. `http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/`.

[15] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[16] M. J. Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2014.

[17] M. J. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, 2016.

[18] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[19] H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI*, 2005.

[20] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[21] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC*, 2014.

[22] Hans-J. Boehm. avoiding thin-air via an operational out-of-order semantics construction. `http://mail.openjdk.java.net/pipermail/jmm-dev/2014-April.txt`, April 2014. jmm-dev mailing list.

[23] G. Boudol and G. Petri. A theory of speculative computation. In *ESOP*, 2010.

[24] S. Castellan. A denotational model for TSO based on event structures. unpublished.

[25] S. Castellan. Weak memory models using event structures. In *JFLA*, 2016.

[26] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, 2007.

[27] S. Chakraborty and V. Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *CGO*, pages 100–110, 2017.

[28] W. W. Collier. *Reasoning about parallel architectures*. Prentice Hall, 1992.

[29] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *POPL*, 2013.

[30] P. Enslow. Multiprocessor organization — a survey. In *ACM Computing Surveys*, 1977.

[31] X. Feng. An operational approach to happens-before memory model, 2015.

[32] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL*, 2016.

[33] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *POPL*, 2017.

[34] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and experience*, 30(11):1203–1233, 2000.

[35] J. Gosling, B. Joy, and G. L. Steele. Java language specification, 1996.

[36] K. E. Gray, G. Kerneis, D. P. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *MICRO*, 2015.

[37] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[38] IBM. Power ISA version 2.06, 2009.

[39] SPARC International. *The SPARC architecture manual (version 9)*. Prentice-Hall, 1994.

[40] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *ESOP*, 2010.

[41] A. S. A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. In *LICS*, 2016.

[42] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, 2017.

[43] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL*, 2016.

[44] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. 2017.

[45] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9), 1979.

[46] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, 2012.

[47] Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. 2017.

[48] J. Manson. `http://jeremymanson.blogspot.co.uk/2007/05/roach-motels-and-java-memory-model.html`, May 2007. Jeremy Manson's blog.

[49] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *POPL*, 2005.

[50] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, 2012. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`.

[51] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an sc-preserving compiler. In *PLDI*, 2011.

[52] P. E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* January 2015.

[53] P. E. McKenney. RCU Linux usage, 2016. `http://www.rdrop.com/users/paulmck/RCU/linuxusage.html`.

[54] P. E. McKenney, T. Riegel, and J. Preshing. N4036: Towards implementation and use of memory order consume, 2014.

[55] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, and O. Giroux. N4321: Towards implementation and use of memory order consume, 2015.

[56] P. E. McKenney and R. Silvera. Example POWER implementation for C/C++ memory model. `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2009.02.27a.html`, February 2009. ISO/IEC JTC1 SC22 WG21 N2745 = 08-0255 - 2008-08-22 (REVISED).

[57] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N.M. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. In *PLDI*, 2016.

[58] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, pages 187–196, 2013.

[59] R. Morisset and F. Zappa Nardelli. Partially redundant fence elimination for x86, arm, and power processors. In *ICCC*, pages 1–10, 2017.

[60] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *ICFP*, 2014.

[61] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In *Proceedings of Semantics of Concurrent Computation*, 1979.

[62] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency. In *OOPSLA*, 2016.

[63] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[64] G. Petri, J. Vitek, and S. Jagannathan. Cooking the books: Formalizing JMM implementation recipes. In *ECOOP*, 2015.

[65] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL*, 2016.

[66] A. Podkopaev, O. Lahav, and V. Vafeiadis. Promising compilation to ARMv8 POP. In *ECOOP*, pages 22:1–22:28, 2017.

[67] A. Podkopaev, I. Sergey, and A. Nanevski. Operational aspects of C/C++ concurrency. In *CoRR*, 2016.

[68] W. Pugh. Causality test cases, 2004. available at `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`.

[69] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In *In submission*.

[70] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPOPP*, 2007.

[71] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, 2012.

[72] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, June 2011.

[73] S. Sarkar, P. Sewell, L. Maranget, P. Pawan, and F. Zappa Nardelli. PPCMem 1. `https://www.cl.cam.ac.uk/~pes20/ppcmem/`.

[74] J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[75] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[76] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), 2013.

[77] P. Sewell. C/C++11 mappings to processors. `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`.

[78] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *ISCA*, 2012.

[79] G. Steele. `https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg00404.html`, December 2001. ll1-discuss mailing list.

[80] A Terekhov. Brief tentative example x86 implementation for C/C++ memory model. `http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001933.html`, December 2008. cpp-threads mailing list.

[81] L. Torvalds. Re: [RFC][PATCH 0/5] arch: atomic rework, February 2014. `https://gcc.gnu.org/ml/gcc/2014-02/msg00384.html`.

[82] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

[83] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, 2015.

[84] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[85] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *SAS*, pages 146–162, 2011.

[86] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[87] K. Walfridsson. `https://kristerw.blogspot.co.uk/2017/06/fipa-pta.html`, June 2017. Krister Walfridsson's blog.

[88] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.

[89] J. Wickerson and M. Batty. Taming the complexities of the C11 and OpenCL memory models. *CoRR*, 2015.

[90] Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. An operational framework for specifying memory models using instantaneous instruction execution. *CoRR*, abs/1705.06158, 2017.

[91] Y. Zhang and X. Feng. An operational approach to happens-before memory model. In *TASE*, 2013.