THÈSE DE DOCTORAT
de l'Université de recherche Paris Sciences Lettres
PSL Research University

Préparée à l'École normale supérieure

Compiler Optimisations and Relaxed Memory Consistency Models

ED 386 SCIENCES MATHÉMATIQUES DE PARIS CENTRE
Spécialité INFORMATIQUE

Robin Morisset
Version du 12/07/2017

Dirigée par Francesco Zappa Nardelli

COMPOSITION DU JURY :

M. Andrew Appel
Princeton University
Rapporteur (non membre du jury)

M. Doug Lea
State University of New York
Rapporteur

M. Mark Batty
University of Kent
Examinateur

M. Derek Dreyer
Max Planck Institute for Software Systems
Examinateur

M. Marc Pouzet
École normale supérieure
Président du jury

M. Dmitry Vyukov
Google
Examinateur

M. Francesco Zappa Nardelli
Inria
Directeur de thèse

ENS ÉCOLE NORMALE SUPÉRIEURE
PSL RESEARCH UNIVERSITY PARIS

Acknowledgments

First, I would like to thank Francesco for being the best advisor I could have hoped for, leaving me great freedom in what to research while always being available when I needed advice. I hope to have learned some of his riguor and pedagogy through these years.

I am also grateful to all the members of my jury for their participation, and in particular to Andrew and Doug for having reviewed very thoroughly this thesis in such a short time.

I was supported throughout this work by a fellowship from Google for which I am thankful. Google, and J.F. Bastien in particular also offered me the wonderful opportunity to apply my research in the Clang compiler through a 3 month internship that I greatly appreciated.

Throughout the rest of my thesis I frequently interacted with the other members of the Parkas team, who made it a very welcoming place to work. I would especially like to thank Guillaume Baudart, Adrien Guatto and Nhat Minh Lê for the many coffee breaks we shared discussing everything and anything, and for the sleepless nights I spent writing papers with Adrien and Nhat.

Obviously, research is frequently a group effort, and I would like to thank now all those I collaborated with over the past few years. Obviously this includes Francesco, Adrien, and Nhat, but also Thibaut Balabonski, Pankaj Pawan, and Viktor Vafeiadis. I also benefited from more informal research collaborations and discussions, in particular with Peter Sewell's team in Cambridge: Mark Batty, Kayvan Memarian, Jean Pichon, Peter Sewell and others made each of my trips to Cambridge remarkably productive and enjoyable.

I am also deeply thankful to Jean-Christophe Filliâtre. He not only made me discover research on compilers through his course in 2009 and his recommendation for my first internship; he also made me a TA in 2015 for his compilation course. This teaching work was a pleasure, with wonderful students, and gave me something to look forward to every week, even when nothing was working on the research side.

I would also like to thank my family, for believing in me even when I didn't and supporting me all these years. I wouldn't be here without you.

Last but not least, I would like to thank all of my friends, who made these years bearable and often even fun.

Contents

Chapter 1

Why study compiler optimisations of shared memory programs?

Multicore computers have become ubiquitous, in servers, personal computers and even phones. The lowest abstraction for programming them is the *shared-memory, multi-threaded program*, in which several sequences of instructions (threads) are executed concurrently, accessing the same data. When these programs are compiled, it is usually done piecemeal: the compiler only see fragments of a thread at any time, unaware of the existence of other threads. This disconnect can lead compilers to break the expectations of the programmers regarding such multi-threaded programs. Consider for example the following program which shows two threads executing side-by-side:

Initially, `x = y = 0`

Thread 1	Thread 2
`x:= 1;`	`if (x > 0) {`
`if (y > 0) {`	`  x := 0;`
`  print x;`	`  y := 1;`
`}`	`}`

It has two global variables, `x` and `y` which are initialized at 0, and they are shared by the two threads. We assume both threads are started at the same time. What values can the first thread print? At first glance, 0 is the only value that can be printed: if the conditional in the left thread is taken, it means the store of 1 to `y` has happened, and so the store of 0 to `x` has happened too. Since it is also in a conditional, it must have happened after the store of 1 to `x`. So the value of `x` is 0 when it is printed.

But if the first thread is compiled by an optimising compiler, the compiler will execute the constant propagation optimisation: since there is no visible access to `x` between its store and its load, the compiler will assume the value has not changed and will replace `print x` by `print 1`, introducing a new behaviour.

Is this a bug in the compiler? No, the problem was our assumption that all threads have the same view of memory at all times, and that executing a multi-threaded program is equivalent to repeatedly picking one thread and executing it for one step. These assumptions are called the *Sequential Consistency model* (SC) and were first formalized in [Lam79]. Constant propagation is not the only optimisation that can break the SC model. Common Sub-expression Elimination (CSE), Loop Invariant Code Motion (LICM), Global Value Numbering (GVN) and Partial Redundancy Elimination (PRE)

are some other examples. Consequently, most languages have decided to offer weaker guarantees to the programmer, to preserve the compilers ability to perform these optimisations.

Even if a compiler were to forsake all SC-breaking optimisations, programmers still could not rely on the SC model, because of details in the architecture of the underlying processor. Consider for example the so-called *Store Buffering* pattern in Figure 1.1, where one thread writes to a variable and reads from another one, and another thread writes to the second variable and reads from the first one.

Initially, `x = y = 0`

```
x := 1;          y := 1;
print y;         print x;
```

Figure 1.1: Store Buffering (SB) pattern

In an SC world, the outcomes of such a program can be listed by enumerating all possible interleavings of the threads (a more efficient way will be described later in the section on axiomatic memory models, see Section 2.2.2). Such an enumeration can easily prove that in any execution, at least one of the two print operations will print 1.

Yet, when implemented directly in assembly, this program will occasionally print two 0s on most multicore/multiprocessor platforms (including x86, ARM, Power, Sparc, Alpha). This can be explained by the presence of store buffers in these processors, so that they do not have to wait for stores to go all the way to memory. Loads check the local store buffer for stores to the same address so single-threaded programs do not notice this optimization, but it is observable to multi-threaded programs. In this instance for example, both stores can go into the store buffers, then both loads will check their own store buffer, not see any store to the same address, and load the value 0 from their cache (or directly from main memory). Only then will the stores be propagated to the memory. In practice, processors do not need to have physical store buffers to exhibit this behaviour, it can emerge from the coherency protocol of normal caches. In order to enforce the SC behaviour, it is necessary to include a special instruction between the stores and the loads, such as `mfence` on x86. All such instructions whose only purpose is to limit the observable effect of processor optimisations are called *fences*, and are usually quite slow.

As a consequence, programming languages that allow writing multi-threaded programs must carefully define what guarantees they offer the programmer. The compiler will then be able to both limit its optimisations and insert fence instructions in the right places to enforce these guarantees. Such a set of guarantees offered to the programmer by a platform is called a *memory model* (occasionally "memory consistency model" in the literature). The C and C++ standards committees have added such a memory model to these languages in 2011 (that we call the C11 model in the rest of this thesis). The C11 model is based on two ideas:

1. The compiler will ensure that code where no memory location is accessed concurrently behaves like in the SC model.

2. Otherwise, the programmer must add annotations to the loads and stores to describe what semantics they want from them. These annotated loads and stores are called *atomic operations* or just atomics, and the annotation is called a memory order

or attribute. Obviously, the stronger the semantics wanted, the more careful the compiler must be.

This model is convenient in that compilers can optimise code without concurrency primitives like in the SC model. They only have to be careful when encountering either locks or atomic operations: these force them to disable some of their own optimisations, and to introduce fence instructions to block processor optimisations.

Contributions In this thesis we present work on the optimisation of shared-memory concurrent programs, with a focus on the C11 memory model.

- The C11 model is highly complex. It has been formalized, but it was originally designed by the standards committee based on what seemed reasonable to implement at the time. It is natural to wonder whether it respects some basic intuitions: is it sound to replace a relaxed atomic (that is supposed to have the weakest guarantees) by a stronger one (monotonicity)? Is it sound to decompose a complex computation into several simpler ones with temporary variables? We answer these questions and others by the negative in Chapter 3, and propose modifications to the model that fix some of these warts.

- In Chapter 4, we study which traditional compiler optimisations designed in a sequential world remain correct in the C11 model. As expected, they remain correct on code without atomics, but must be very careful when modifying code containing atomics. We prove the correctness of criteria under which they remain sound.

- Armed with this theory of correct optimisations, we built a tool that uses random testing to find mistakes in the implementation of the C11 memory model in GCC. This tool is presented in Chapter 5.

- Another important question related to the C11 model is how usable it is in practice for writing complex lock-free algorithms. What patterns are the most frequently used? Can non-trivial code using atomic operations be proven correct? To answer these questions, we designed and implemented a dynamic scheduler for Kahn Process Networks in C11, starting from a naive sequentially consistent algorithm and progressively refining it to use more efficient atomic operations (Chapter 6).

- Finally, we looked at how compilers can optimize the fence instructions that they have to insert around atomic operations to prevent reorderings of instructions by the processor. We propose a novel algorithm for fence elimination based on Partial Redundancy Elimination (Chapter 7).

- Before all of that, we need to provide some background information. In particular, we will see formal models for x86, ARM, Power, and C11 (Chapter 2). This chapter includes some original work in Section 2.3.1, that shows how a compiler optimisation is unsound in the Power model.

Collaborations Much of the work presented in this thesis was done in collaboration with others and resulted in publications:

- The study and testing of the optimisations done by Clang and GCC around atomics was done with Pankaj Pawan and Francesco Zappa Nardelli. It was presented at PLDI 2013 [MPZN13] and is described in Section 4.1 and Chapter 5

- The formal study of optimisations in C11 and its variants was done with Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty and Francesco Zappa Nardelli. It was presented at POPL 2015 [VBC+15] and is described in Chapter 3 and Section 4.2

- LibKPN (Chapter 6) was joint work with Nhat Minh Lê, Adrien Guatto and Albert Cohen. It has not yet been published.

- The fence elimination algorithm presented in Chapter 7 was designed during an internship at Google under the supervision of Jean-François Bastien and evaluated later with Francesco Zappa Nardelli. It will be presented at CC 2017 [MZN17].

Chapter 2

A primer on memory models

2.1 Examples of non-sequentially consistent behaviors

We already saw in the introduction the SB pattern (Figure 1.1), and how it can be caused by First-In First-Out (FIFO) store buffers in each core.

This is not the only processor optimisation that can be observed by multi-threaded programs. Consider for example the program in Figure 2.1, called *Message Passing* or MP in the literature:

Initially, `x = y = 0`

```
x := 1;              if (y == 1)
y := 1;                   print x;
```

Figure 2.1: Message Passing (MP) pattern

According to the SC model, this pattern should never print 0. And indeed it never does on x86, as the store buffers behave in a First-In First-Out (FIFO) way, so by the time the store of y is visible to the second thread, so is the store of x. But on ARM and Power, it will occasionally print 0. Several architectural reasons can explain this. For example, it could be that the store buffers are not FIFO, letting the store of y reach global memory before the store of x. Or this behaviour could be caused by speculation: the second core may guess that the branch will be taken, and execute the load of x in advance, possibly even before the load of y.

Another example is the *Load Buffering* (LB) pattern in Figure 2.2. Like the previous one, this program cannot print two ones in the SC model or on x86 processors, but it can on ARM and Power processors. This can be caused by the processors executing the

Initially, `x = y = 0`

```
print y;             print x;
x := 1;              y := 1;
```

Figure 2.2: Load Buffering (LB) pattern

instructions out-of-order and deciding to execute the stores before the loads.

An even more surprising example is the *Independent Reads of Independent Writes* (IRIW) pattern in Figure 2.3.

Initially, `x = y = 0`

`x := 1;`	`print x;` `print y;`	`print y;` `print x;`	`y := 1;`

Figure 2.3: Independent Reads of Independent Writes (IRIW) pattern

This is yet another case where x86 follows the SC model, but ARM and Power do not. More precisely, different cores can see the stores to x and y happening in different orders, resulting in both central threads printing 1 and then 0. This can happen if the L2 cache is shared between the threads 0 and 1, and is shared between the threads 2 and 3: the store of x will be visible first to the thread 1, and the store of y will be visible first to the thread 2.

Cache coherency is usually defined as all cores agreeing on the order in which all stores to each individual memory location reach the memory. While the behaviour shown above is quite unintuitive, it is not a violation of cache coherency: it is only the order of stores to *different* locations that is not well defined.

2.2 Hardware memory models

Given the wide range of surprising behaviours shown in Section 2.1 it is necessary for any programmer writing concurrent code in assembly to have a model of the observable effects of these processor optimisations. Processor vendors offer informal specifications in their programming manuals, but these are often ambiguous and sometimes outright incorrect. This led to a significant amount of research in formal memory models for hardware architectures.

2.2.1 The x86 architecture

As we saw in Section 2.1, x86 differs from sequential consistency mostly by its behaviour in the SB pattern. It is possible to limit the behaviour in this pattern to those admissible under SC by inserting a special instruction called `mfence` between the stores and the loads. x86 also has a LOCK prefix that can be added to some instructions to make them atomic.

The x86 architecture is one of the first whose memory model was rigorously formalized and validated against the hardware. The resulting model is called x86-TSO and is described in [OSS09, SSO$^+$10]. This model was formalized in two different ways, an operational and an axiomatic models. We describe both in more details below, as an example of each kind of model.

2.2.1.1 The x86 operational model

The first model is an *operational model* [BP09]: it is the description of an abstract machine that simulates an x86 processor and explicitly models the store buffers and a global lock.

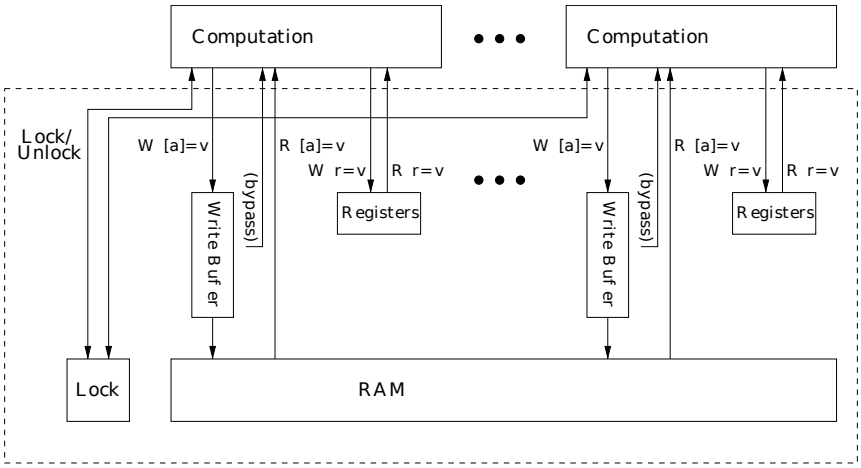


Figure 2.4: Abstract machine for x86-TSO from [OSS09]

Such a model is convenient for building an intuition, and can easily show that a behaviour is possible. Unfortunately, showing a behaviour to be impossible can be difficult as all transitions must be explored.

In this view of x86-TSO, each processor proceeds mostly independently and completely sequentially, communicating with its store buffer, the global lock and the memory according to the following rules (see Figure 2.4):

- Whenever it executes a store, the store is sent to a local store buffer that acts as a First-In First-Out (FIFO) queue.
- Whenever it executes a load, it first checks if there is a store to the same address in the local store buffer. If there is, it returns the value of the latest such store. Otherwise it asks the global memory for the value of this memory location. In both cases, the load is only executed if the global lock is not taken by another processor.
- Whenever it executes an instruction with the LOCK prefix, it first flushes its local store buffer to memory, then takes the global lock, executes the instruction, flushes the buffer again, and release the lock afterwards. In x86, instructions with the LOCK prefix are executed atomically, and are used to implement primitives like *fetch-and-add* or *compare-and-swap*.
- Whenever it executes a `mfence` instruction, it flushes its local buffer.
- It can flush the latest store(s) from its buffer to the shared memory at any point where no other processor holds the global lock.

Consider again the SB example (Figure 1.1). Without fences, the following behaviour is permitted by the model:

1. The left thread sends the store to `x` to its store buffer.
2. It then executes the load of `y`. Since there is no store to `y` in its store buffer, it reads the initial value from global memory and prints 0.
3. The right thread sends the store to `y` to its store buffer.

4. The right thread executes the load of `x`. Since the store of `x` is only in the left thread's core's store buffer, it reads the initial value of `x` from global memory and prints 0.

With the `mfence` instruction added between every store and load, this execution would be impossible: the left thread would have to flush its store buffer to memory before executing the load. When the right thread later reads from memory, it would see the correct value and print 1.

However, note that in this model the only way to check that a behaviour can not happen (for example both thread printing 0) is to explore every transition of the system exhaustively.

2.2.1.2 The x86 axiomatic model

The second model of x86-TSO is an *axiomatic model*: an execution is valid and can happen if there exists a set of relations between the memory accesses of the execution that verifies some set of axioms. The model is entirely defined by its relations and axioms.

Axiomatic models are usually less intuitive than operational ones, but easier to use for verification purposes [AKT13], as they simplify proofs that a behaviour is impossible.

Here is how the axiomatic formalisation of x86-TSO works in more detail. For each thread, the set of memory events (store, loads, locked operations) this thread can produce is generated, assuming completely arbitrary values returned for each read. The data of such a set and the program order relation (po) is called an event structure in [OSS09], but we will call it a *event set* here, as the term event structure has come to mean something else in more recent papers.

An event set is a valid execution if there exists an *execution witness* made of two relations *reads-from (rf)* and *memory-order (mo)*[1] which respects the following axioms:

1. rf is a function from loads to store, and if $w = \mathsf{rf}(r)$, then w and r share the same value and memory location;
2. mo is a partial order on memory events that is a total order on the stores to each memory location (but does not necessarily relate stores to different memory locations);
3. if $w = \mathsf{rf}(r)$, then there is no other write w_2 to the same location with $w <_{\mathsf{mo}} w_2$ and $w_2 <_{\mathsf{mo}} r$ or $w_2 <_{\mathsf{po}} r$;
4. program order is included in memory order for all of the following cases:
   - the first event is a read,
   - both events are writes,
   - the first event is a write, the second is a read, and there is a `mfence` instruction in-between,
   - one of the event comes from a locked instruction;
5. all memory events associated to a given locked instruction are adjacent in memory order.

[1] The original model also includes an initial_state function that we ignore for simplicity

Consider again the SB example (Figure 1.1). It has a very large number of event sets, as we must generate one for every value that each load could read, for example those in Figure 2.5:

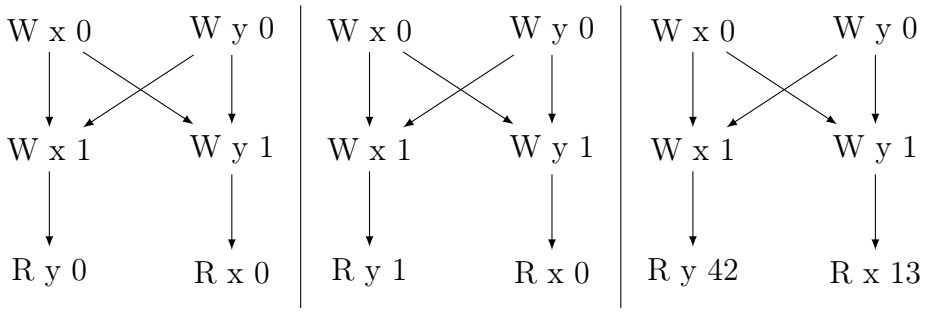


Figure 2.5: Three examples (out of many) of event sets of SB. Note the arbitrary values returned by reads.

For each of these we can try building mo and rf. In particular, to check that SB can print two 0, we must build these two relations for the corresponding event set. We clearly can, as shown in Figure 2.6 where all the axioms are respected.

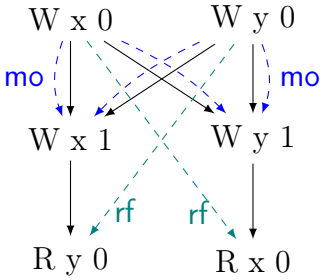


Figure 2.6: building mo and rf for the anomalous SB result

This might not look like an improvement over the operational model. But it becomes useful when we try to show that some behaviour *cannot* happen. For instance, we can now easily show that SB cannot print two 0 with a `mfence` between the store and the load in each thread. By the axiom 4, the events in each thread are necessarily ordered by mo. And they are mo-after the initial stores. We assume the existence of the initial stores that are po-before all other events to avoid having to special-case them. By the Axiom 1, each read must be related by rf to the initial stores. But that is in contradiction with Axiom 3: we have successfully proven that adding a `mfence` between the stores and the loads prevent this program from printing two 0.

A complete description of the models can be found in the aforementioned papers.

2.2.2 The Power and ARM architectures

A memory model is said to be stronger than another one if it allows less behaviours. SC is clearly the strongest of all the memory models we saw. As we saw earlier with the MP, LB and IRIW examples, both Power and ARM have significantly weaker memory models than x86.

These models are highly subtle and poorly documented, which led to a list of progressively more complete and readable formalisations in the literature [AFI+09, AMSS10, SSA+11, SMO+12, MMS+12, AMT14]. We will only present in detail the formalization in [AMT14], which we rely upon in Chapter 7. This model [2] is axiomatic, and we call it the "Herding Cats" model after the title of the paper.

This formalisation is actually a framework with several parameters that can be instantiated to give a model for Power, ARM, x86 or even SC.

In this framework the semantics of a program is defined in three steps, as in the x86 axiomatic model.

First the program is mapped to a set of memory events (e.g. atomic read or writes of shared memory locations) and several relations among these events. The intuition is that these events capture all possible memory accesses performed by the various threads running in an arbitrary concurrent context. The additional relations capture how several "syntactic" aspects of the source program lift to, and relate, events. These include the *program order* relation (po), that lifts the program order to the events, and the addr, data, and ctrl relations, that lift address, data and control dependencies between instructions to events. This corresponds to the event sets in the model of Section 2.2.1.2

Second, *candidate executions* are built out of the events by imposing restrictions on which values read events can return, each execution capturing a particular data-flow of the program. The restriction are expressed via two relations, called *reads-from (rf)* and *coherence-order (co)*. The former associates each read even to the write event it observes. The second imposes a per-location total order on memory writes, capturing the base property ensured by cache coherence. These are subject to two well-formedness conditions:

- rf is a function from loads to store, and if $w = \mathsf{rf}(r)$, then w and r share the same value and memory location;
- co is a partial order on memory events that is a total order on the stores to each memory location (but does not necessarily relate stores to different memory locations).

Several auxiliary relation are computed out of rf and co. The most important is the relation *from-read* (fr), defined as $\mathsf{rf}^{-1}; \mathsf{co}$, that relates a read to any store that is co-after the one it reads from. An example of that can be seen in Figure 2.7: because the load of x reads from the first store, it is related by fr to every later store to x.

We also compute the projections of rf, fr and co to the events internal to a thread (the related events belong to the same thread, suffix i) or external (the related events belong to different threads, suffix e) to a thread; these are denoted rfi, rfe, fri fre and coe. The construction of these relations corresponds to the construction of rf and mo in the model of Section 2.2.1.2, with co corresponding to mo.

Third, a *constraint specification* decides if a candidate executions are valid or not. HerdingCats is a framework that can be instantiated to different architectures. For this it is parametric in two relations *happens-before* (hb), defining the relative ordering of

[2] This model does not cover mixed-size accesses, nor special instructions like load-linked/store-conditional

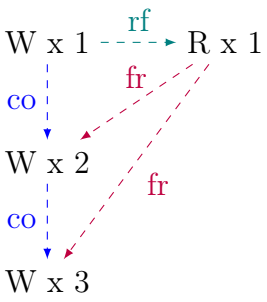


Figure 2.7: co, rf, and fr

the events., and *propagation order* (prop), capturing the relative ordering of memory writes to different locations that somehow synchronise. We detail the instantiation of the framework to x86, Power, and ARM below. The framework then imposes only four axioms on candidate executions to determine their validity. These axioms play the same role as the axioms of the model of Section 2.2.1.2

The first axiom, **SC PER LOCATION**, captures cache coherence: for every memory location considered in isolation, the model behaves as if it was sequentially consistent. In formal terms:

$$\mathsf{acyclic}(\mathsf{poloc} \cup \mathsf{com})$$

where poloc is the program order restricted per location, and com (communication order) is the union of the coherency order co, the reads-from relation rf, and the from-reads relation fr.

The second axiom, **NO THIN AIR**, prevents the so-called *out-of-thin-air reads*, where the read of a value is justified by a store that is itself justified by the original read. Such causality loops are not observable on any modern hardware. In Section 3.1.2 we will see how troublesome a model without that rule is. In formal terms, the rule forbids cycles in *happens-before* (hb), which is the first parameter of the model:

$$\mathsf{acyclic}(\mathsf{hb})$$

The third axiom, **OBSERVATION**, mostly enforces that loads cannot read from stores that have been overwritten in-between. In particular, it is what makes the MP(Figure 2.1) pattern work in x86, and with the right fences in ARM and Power too.

$$\mathsf{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*)$$

In this rule fre is the restriction of fr to an inter-thread relation, hb^* is the transitive and reflexive closure of hb, and prop is the second parameter of the model. The stated justification in the paper for prop is to relate stores to different memory locations that are synchronised somehow, and thus represent *cumulativity*. In the words of the paper, "a fence has a cumulative effect when it ensures a propagation order not only between writes on the fencing thread (i.e. the thread executing the fence), but also between certain writes coming from threads other than the fencing thread." [AMT14, p. 18].

The fourth axiom, **PROPAGATION**, enforces the compatibility of the coherence order with this prop relation:

$$\mathsf{acyclic}(\mathsf{co} \cup \mathsf{prop})$$

In fact, prop is not entirely limited to stores on Power (see below), and has a term that can relate loads. It is this term in combination with the propagation axiom that bans the IRIW pattern (Figure 2.3) when the loads are separated by heavyweight fences.

2.2.2.1 Instantiations of the framework

The model presented above is completely generic over hb and prop. We will now show what definition to give to these two relations to make the model describe different architectures.

Instantiation for SC Getting back sequential consistency in this framework is easy: just define $\mathsf{hb} = \mathsf{po} \cup \mathsf{rfe}$ and $\mathsf{prop} = \mathsf{po} \cup rf \cup \mathsf{fr}$ (where rfe is the restriction of rf to an inter-thread relation). Then the first three axioms are implied by the fourth one which is equivalent to $acyclic(\mathsf{po} \cup \mathsf{com})$, which is a well-known characterisation of SC [Lam79].

Instantiation for x86 Similarily, x86-TSO can be recovered by defining $\mathsf{hb} = \mathsf{po} \backslash \mathsf{WR} \cup \mathsf{mfence} \cup \mathsf{rfe}$ and $\mathsf{prop} = \mathsf{hb} \cup \mathsf{fr}$, with mfence being the restriction of po to pairs of accesses with a *mfence* instruction in between, and $\mathsf{po} \backslash \mathsf{WR}$ the program order relation from which every pair from a write to a read has been removed. In this instantiation, the axioms NO THIN AIR and OBSERVATION are implied by PROPAGATION.

Instantiation for Power The Power memory model is significantly more complex. Here are the definitions of hb and prop:

$$
\begin{aligned}
\mathsf{fences} &\stackrel{\mathsf{def}}{=} (\mathsf{lwsync} \backslash \mathsf{WR}) \cup \mathsf{hwsync} \\
\mathsf{propbase} &\stackrel{\mathsf{def}}{=} \mathsf{rfe}?; \mathsf{fences}; \mathsf{hb}^* \\
\mathsf{prop} &\stackrel{\mathsf{def}}{=} (\mathsf{propbase} \cap \mathsf{WW}) \cup (\mathsf{com}^*; \mathsf{propbase}^*; \mathsf{hwsync}; \mathsf{hb}^*) \\
\mathsf{dp} &\stackrel{\mathsf{def}}{=} \mathsf{addr} \cup \mathsf{data} \\
\mathsf{rdw} &\stackrel{\mathsf{def}}{=} \mathsf{poloc} \cap (\mathsf{fre}; \mathsf{rfe}) \\
\mathsf{detour} &\stackrel{\mathsf{def}}{=} \mathsf{poloc} \cap (\mathsf{coe}; \mathsf{rfe}) \\
ii_0 &\stackrel{\mathsf{def}}{=} \mathsf{dp} \cup \mathsf{rdw} \cup \mathsf{rfi} \\
ci_0 &\stackrel{\mathsf{def}}{=} (\mathsf{ctrl} + \mathsf{cfence}) \cup \mathsf{detour} \\
ic_0 &\stackrel{\mathsf{def}}{=} \emptyset \\
cc_0 &\stackrel{\mathsf{def}}{=} \mathsf{dp} \cup \mathsf{poloc} \cup \mathsf{ctrl} \cup (\mathsf{addr}; \mathsf{po}) \\
ii &\stackrel{\mathsf{def}}{=} ii_0 \cup ci \cup (ic; ci) \cup (ii; ii) \\
ci &\stackrel{\mathsf{def}}{=} ci_0 \cup (ci; ii) \cup (cc; ci) \\
ic &\stackrel{\mathsf{def}}{=} ic_0 \cup ii \cup cc \cup (ic; cc) \cup (ii; ic) \\
cc &\stackrel{\mathsf{def}}{=} cc_0 \cup ci \cup (ci; ic) \cup (cc; cc) \\
\mathsf{ppo} &\stackrel{\mathsf{def}}{=} (ii \cap \mathsf{RR}) \cup (ic \cap \mathsf{RW}) \\
\mathsf{hb} &\stackrel{\mathsf{def}}{=} \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}
\end{aligned}
$$

In these, lwsync and hwsync are restrictions of po to pairs of accesses with respectively a *lwsync* or *hwsync* (two kinds of fence instruction that exist on Power, respectively lightweight and heavyweight synchronisation) in between. ctrl, addr and data are respectively control, address and data dependencies. Finally, ctrl + cfence refers to control dependencies which have been reinforced by an *isync* fence.

The relations ii, ic, ci cc are defined as the least fixed point of the equations above. The intuition behind these is that i stands for "initiation" of an access, while c stands for "commit" of an access; which is similar to older models for Power [MMS+12].

Instantiation for the ARM architecture The instantiation of HerdingCats for ARM follows the IBM Power one, with three differences:

- the relation hwsync is replaced by the relation dmb ish, that is the restriction of po to pairs of accesses with respectively a `dmb ish` synchronising barrier in between; similarly the relation cfence is replaced by the relation isb, that is the restriction of po to pairs of accesses with respectively a `isb` instruction in between;
- the relation lwsync is removed;
- the relation cc_0 is redefined as $\mathsf{dp} \cup \mathsf{ctrl} \cup (\mathsf{addr}; \mathsf{po})$, losing the poloc term (the reason for this is detailed at length in [AMT14]).

2.2.3 Other architectures

Several other architectures have complex and documented memory models: Alpha [alp98], Itanium [ita02], Sparc [WG03]. Because these architectures are not as popular as x86 and ARM, we do not consider them in the rest of this thesis. We also do not treat GPUs, whose memory model has only started to be studied over the past few years.

I would like to mention one point about Alpha however: it is an example of how weak a hardware memory model can be. In particular it is the only architecture that does not respect address dependency by default between reads: the pseudo-code (using the C notation for pointers) in Figure 2.8 can print 0:

Initially, `x = 0`, `z = 42` and `y = &z`

```
x := 1;                          r = y;
heaviest_fence()                 print *r;
y := &x;
```

Figure 2.8: Example of an address dependency

It is as if the load of `*y` was reordered before the load of `y`. This could be caused by using independent caches for different cache lines, and in particular for `y` and `x`.

2.3 Language memory models

As we saw in Chapter 1, languages must also have a memory model that describes both how the optimisations done by the compiler can change the semantics of concurrent programs and what fences it introduces to restrict the underlying processor from doing the same.

Initially, x = y = z = 0

z := 42;
z := 13;
sync;
x := 42;
	r0 = z;
	print r0; // 42
	r1 = x;
	print r1; // 42
	if (r0 == r1)
	z := r1;
	sync;
	y := 1;
		print y; // 1
		sync;
		print z; // 13

Figure 2.9: WaR elimination in the Power model: counterexample with a dependency

2.3.1 Compiler optimisations in the Power model?

Since hardware memory models are reasonably well understood, it is tempting to lift one of them to the language level. We want a weak one, to allow many compiler optimisations. Since Power has one of the weakest hardware models that has been extensively studied, it is the obvious pick.

In this section we show why it is not a good option: some common compiler optimizations are unsound in the Power memory model[3].

In particular, one of the optimizations we study in Chapter 4 is called *Write-after-Read elimination* (WaR) in the literature, and consists in eliminating a store to memory when the value stored has just been read.

Figure 2.9 provides a counterexample to the claim that WaR elimination is sound in the Power model: it is impossible to have all the print instructions give the expected result before the highlighted store (in bold and red) is removed and it becomes possible afterwards. Compilers could try to remove this store because it is only executed when the value to be stored is equal to the value that was just read, so it seems unable to modify the state of memory. The synchronisation on `x` makes the stores to `z` to happen before the highlighted store. That store, in turn, happens before the load of `z` in the last thread. As a result, this load can not see the store of 13, as it is overwritten by the highlighted store with the value 42. If that store is removed, the printing of 13 suddenly becomes possible.

It might seem that this example relies on the dependency between the load and the store, and that we could find some local criterion under which WaR elimination is valid on Power, such as an absence of dependency between the load and the store.

Figure 2.10 proves that there is no local condition sufficient to make WaR elimination sound: even when the store to be eliminated is adjacent to the load that justifies it, it is still unsound to remove it. Like in the previous example, the print operations can only give the results in the comments if the highlighted access is removed, so removing it is unsound. This counterexample was carefully checked both by hand and by an unreleased software tool by Luc Maranget. Here is some intuition for how it works. The synchronisation on `y` between the last two threads ensures that the to-be-eliminated access is visible to the last load of `z` in the last thread. The complex synchronisation on `x` in the first 4 threads ensures that this second store of 42 in `z` is after the store of 13 in the coherency order.

[3]This section presents original work, by opposition to the rest of this chapter that only provides background.

Initially, $x = y = z = 0$

`z := 42;`
`z := 13;`
`lwsync;`
`x := 2;`
	`x := 1;`	`print x; // 2`
		`lwsync;`
		`print x; // 1`
			`print z; // 42`
			`lwsync;`
			`print x; // 0`
			`print x; // 1`
			`lwsync;`
			`r = z;`
			`print r; // 42`
			`z = r;`
			`lwsync;`
			`y := 1;`
				`print z; // 13`
				`print y; // 1`
				`lwsync;`
				`print z; // 13`

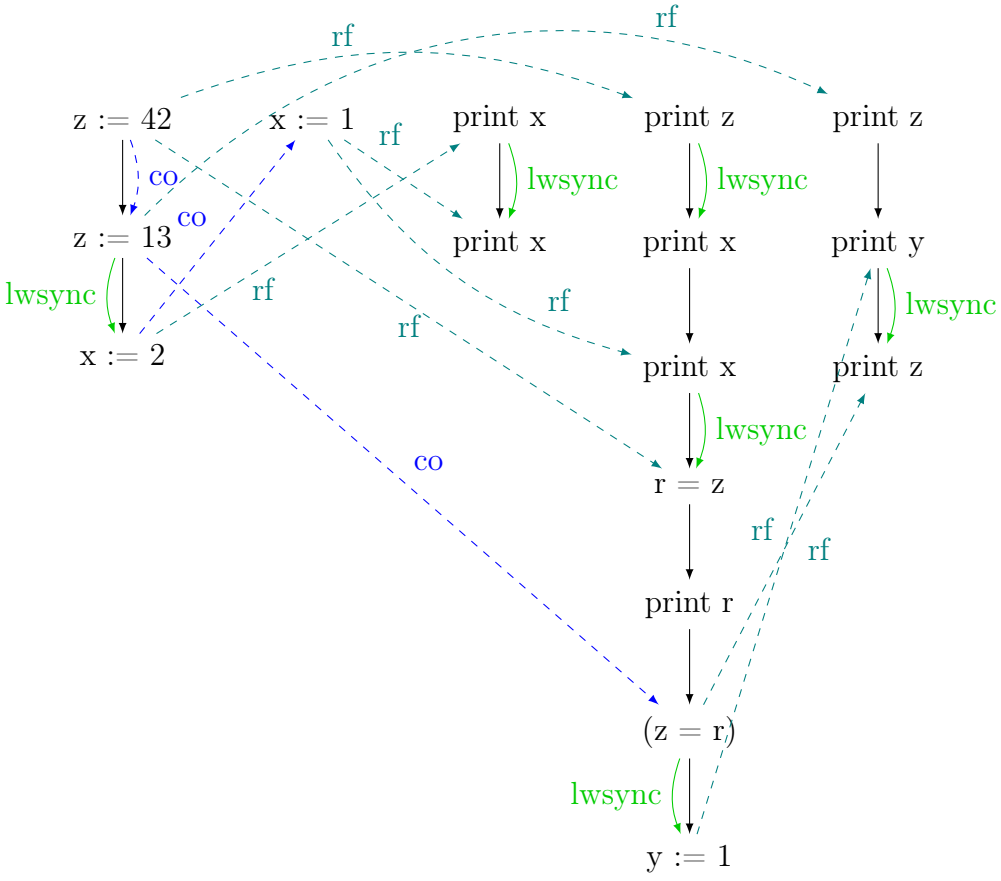


Figure 2.10: WaR elimination in the Power model: counterexample in the most general case

We use this complex pattern instead of a simpler message-passing pattern to allow the last load of 42 in the second to last thread.

2.3.2 Data-Race Freedom (DRF)

A fairly simple and popular language-level memory model is Data-Race Freedom (DRF) [AH90].

A *data-race* is described as two memory events, one of which at least is a write, accessing the same memory location concurrently. DRF states that every program that is free from data-races on all of its executions (and in particular any program that protects all of its concurrently accessed locations by locks) behaves as it would in the SC model. Any data-race is undefined behaviour, and any program that can exhibit one has no defined semantics.

This model is attractive for three reasons:

- it is rather easy to explain to programmers;
- common compiler optimisations remain correct on any piece of code without locks [Sev11, Sev08];
- fence instructions, that limit the optimisations performed by the processor, only have to be introduced inside the lock and unlock methods.

However, the fact that this model gives no defined semantics to racy code is a major weakness. It makes it unsuitable for use by both Java and C/C++, although for different reasons.

It is unsuitable to Java, because racy code can break safety invariants (since by definition it can do absolutely anything). This is unacceptable for Java, as no valid Java code should be able to forge a pointer to the internals of the JVM. As a result, the Java community had to develop a much more complex memory model [MPA05]. This model evolved over the years as some flaws were discovered (see for example [SA08]). As we are focused on the C11 model in the rest of this thesis, we won't describe it here.

C and C++ cannot use DRF either, because of the second weakness: it is impossible to write low-level racy code in the DRF model, as all concurrent accesses must be protected by some synchronisation. To solve this problem, C and C++ use a variant of DRF, where some accesses can be annotated by the programmer to allow races involving them. Such *atomic accesses* have a behaviour that is complex and dependent on the annotation provided by the programmer. We will now explain this model in more details.

2.3.3 The C11/C++11 model

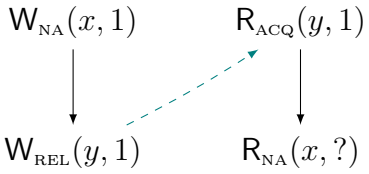
The C11/C++11 memory model was first presented in the C++11 standard [Bec11]. It offers four kind of operations:

- stores;
- loads;
- read-modify-write (RMW) operations like compare-and-swap (CAS) or fetch-and-add;

- fences.

Each store and load is either not annotated and called not-atomic (NA), or annotated with one of the following *memory orders* (also called memory attributes):

- Sequentially consistent (SC): all such accesses are fully ordered. If all concurrent accesses in a program have this attribute, then the program follows the SC model;
- Release (REL, for writes) and Acquire (ACQ, for reads) enforce the SC behavior specifically in the MP pattern. Anything that happens before a release store is visible to anything that happens after an acquire load that reads from that store. For example, in the following figure, the non-atomic load must read the value 1:



- Consume (CONSUME, for reads only) is a weaker form of acquire: it only prevents the MP pattern anomaly if the second read has an address dependency on the first one. It was introduced to the standard to avoid a fence in that specific case on ARM and Power;
- Relaxed (RLX) accesses offer extremely few guarantees, mostly just cache coherency. They are intended not to require hardware fences on any common architecture, and only weakly restrict compiler optimisations.

Similarily, RMW operations can be SC, REL, ACQ, CONSUME, RLX, or Release-Acquire (REL-ACQ) which combine the properties of REL and ACQ.

Fences can also have one of several memory orders:

- REL fences make subsequent relaxed stores behave more like release stores (we will soon see a more rigorous description, this is only the intuition behind them);
- ACQ fences make previous relaxed loads behave more like acquire loads;
- REL-ACQ fences naturally behave like both release and acquire fences;
- SC fences additionnaly prevent the anomalous behavior of the SB pattern.

We will now give the details of the formalisation of C11, that was first given in [BOS+11] and was further extended in [BMO+12].

Like the other axiomatic models we have seen, it is organised in three steps. First it builds sets of events (called opsems below), then it considers possible relations between those events (these set of relations are called witnesses below), and finally it filters all these candidate executions by some axioms.

Opsemsets To abstract from the syntax complexity of the C language, we identify a source program with a set of descriptions of what *actions* it can perform when executed in an arbitrary context.

More precisely, in a source program each thread consists of a sequence of *instructions*. We assume that, for each thread, a thread-local semantics associates to each *instruction instance* zero, one, or more shared memory accesses, which we call *actions*. The actions we consider, ranged over by act, are of the form:

$$\begin{aligned} \Phi ::= {} & \mathsf{skip} \mid \mathsf{W}_{(\text{SC}\mid\text{REL}\mid\text{RLX}\mid\text{NA})}(\ell, v) \mid \mathsf{R}_{(\text{SC}\mid\text{ACQ}\mid\text{RLX}\mid\text{NA}\mid\text{CONSUME})}(\ell, v) \\ & \mid \mathsf{C}_{(\text{SC}\mid\text{REL-ACQ}\mid\text{ACQ}\mid\text{REL}\mid\text{RLX})}(\ell, v, v') \mid \mathsf{F}_{(\text{ACQ}\mid\text{REL}\mid\text{REL-ACQ}\mid\text{SC})} \mid \mathsf{A}(\ell) \\ act ::= {} & tid : \Phi \end{aligned}$$

where ℓ ranges over memory locations, v over values and $tid \in \{1..n\}$ over thread identifiers. We consider atomic and non-atomic loads from (denoted R) and stores to (W) memory, fences (F), read-modify-writes (C), and allocations (A) of memory locations. To simplify the statement of some theorems, we also include a no-op (skip) action. Each action specifies its thread identifier tid, the location ℓ it affects, the value read or written v (when applicable), and the memory-order (written as a subscript, when applicable).

We assume a labelling function, lab, that associates action identifiers (ranged over by $a, b, r, w, \ldots$) to actions. In the drawings we usually omit thread and action identifiers.

We introduce some terminology regarding actions. A *read action* is a load or a read-modify-write (RMW); a *write* is a store or a RMW; a *memory access* is a load, store or RMW. Where applicable, we write $\mathsf{mode}(a)$ for the memory order of an action, $\mathsf{tid}(a)$ for its thread identifier, and $\mathsf{loc}(a)$ for the location accessed. We say an action is *non-atomic* if and only if its memory-order is NA, and *SC-atomic* if and only if it is SC. An *acquire* action has memory-order ACQ or stronger, while a *release* has REL or stronger. The *is stronger* relation, written $\sqsupseteq$: $\mathcal{P}(MO \times MO)$, is defined to be the least reflexive and transitive relation containing SC $\sqsupseteq$ REL-ACQ $\sqsupseteq$ REL $\sqsupseteq$ RLX, and REL-ACQ $\sqsupseteq$ ACQ $\sqsupseteq$ CONSUME $\sqsupseteq$ RLX.

The thread local semantics captures control flow dependencies via the *sequenced-before* (sb) relation, which relates action identifiers of the same thread that follow one another in control flow. We have $sb(a, b)$ if a and b belong to the same thread and a precedes b in the thread's control flow. Even among actions of the same thread, the sequenced-before relation is not necessarily total because the order of evaluation of the arguments of functions, or of the operands of most operators, is underspecified in C and C++.

The thread local semantics also captures thread creation via the *additional-synchronised-with* (asw) relation, that orders all the action identifiers of a thread after the corresponding thread fork (which can be represented by a skip action).

Summarising, the thread local semantics gives each program execution a triple $O = (lab, sb, asw)$, called an *opsem*. As an example, Figure 2.11 depicts one opsem for the program on the left and one for the program on the right. Both opsems correspond to the executions obtained from an initial state where y holds 3, and the environment does not perform any write to the shared variables (each read returns the last value written).

The set of all the opsems of a program is an *opsemset*, denoted by S. We require opsemsets to be *receptive*: S is receptive if, for every opsem O, for every read action r in the opsem O, for all values v' there is an opsem O' in S which only differs from O because the read r returns v' rather than v, and for the actions that occur after r in $sb \cup asw$. Intuitively an opsemset is receptive if it defines a behaviour for each possible value returned by each read.

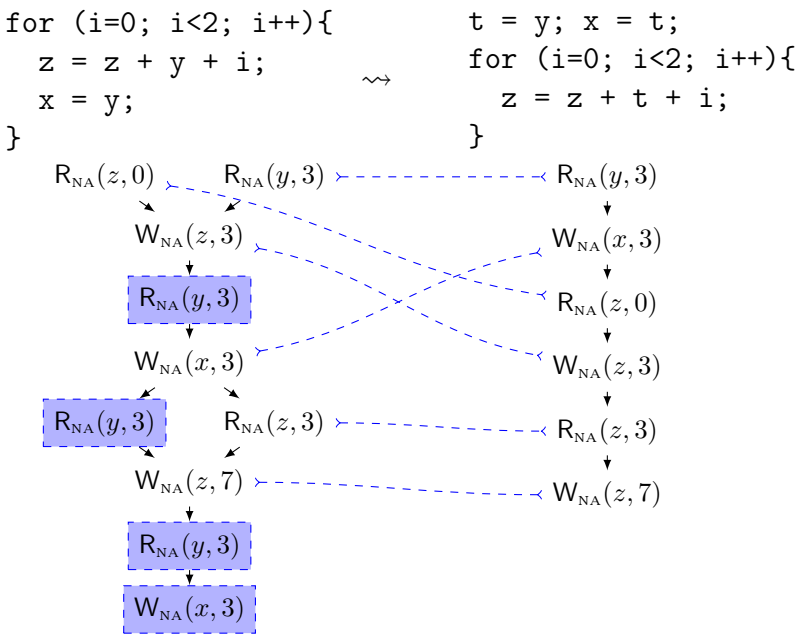


Figure 2.11: Example of the effect of a code transformation (Loop Invariant Code Motion) on an opsem (in which initially $y = 3$ and $z = 0$)

We additionally require opsemsets to be prefix-closed, assuming that a program can halt at any time. Formally, we say that an opsem O' is a prefix of an opsem O if there is an injection of the actions of O' into the actions of O that behaves as the identity on actions, preserves sb and asw, and, for each action $x \in O'$, whenever $x \in O$ and $(sb \cup asw)(y, x)$, it holds that $y \in O'$.

Opsems and opsemsets are subject to several other well-formedness conditions, e.g. atomic accesses must access only atomic locations, which we omit here and can be found in [BOS+11].

Candidate executions The mapping of programs to opsemsets only takes into account the structure of each thread's statements, not the semantics of memory operations. In particular, the values of reads are chosen arbitrarily, without regard for writes that have taken place.

The C11 memory model then filters inconsistent opsems by constructing additional relations and checking the resulting candidate executions against the axioms of the model. We present here a subset of the model, without CONSUME for simplicity. We will see in Section 3.0.1 that consume is effectively unusable with its current definition. For the subset of C11 we consider, a witness W for an opsem O contains the following additional relations:

- The *reads-from* map (rf) maps every read action r to the write action w that wrote the value read by r.
- The *modification-order* (mo) relates writes to the same location; for every location, it is a total order among the writes to that location.
- The *sequential-consistency* order (sc) is a total order over all SC-atomic actions. (The standard calls this relation S.)

From these relations, C11 defines a number of derived relations, the most important of which are: the *synchronizes-with* relation and the *happens-before* order.

$$
\begin{aligned}
\mathsf{isread}_{\ell,v}(a) &\stackrel{\text{def}}{=} \exists X, v'.\ lab(a) \in \{\mathsf{R}_X(\ell, v), \mathsf{C}_X(\ell, v, v')\} \\
\mathsf{isread}_{\ell}(a) &\stackrel{\text{def}}{=} \exists v.\ \mathsf{isread}_{\ell,v}(a) \\
\mathsf{isread}(a) &\stackrel{\text{def}}{=} \exists \ell.\ \mathsf{isread}_{\ell}(a) \\
\mathsf{iswrite}_{\ell,v}(a) &\stackrel{\text{def}}{=} \exists X, v'.\ lab(a) \in \{\mathsf{W}_X(\ell, v), \mathsf{C}_X(\ell, v', v)\} \\
\mathsf{iswrite}_{\ell}(a) &\stackrel{\text{def}}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a) \\
\mathsf{iswrite}(a) &\stackrel{\text{def}}{=} \exists \ell.\ \mathsf{iswrite}_{\ell}(a) \\
\mathsf{isfence}(a) &\stackrel{\text{def}}{=} lab(a) \in \{\mathsf{F}_{\text{ACQ}}, \mathsf{F}_{\text{REL}}, \mathsf{F}_{\text{REL-ACQ}}, \mathsf{F}_{\text{SC}}\} \\
\mathsf{isaccess}(a) &\stackrel{\text{def}}{=} \mathsf{isread}(a) \vee \mathsf{iswrite}(a) \\
\mathsf{isNA}(a) &\stackrel{\text{def}}{=} \mathsf{mode}(a) = \text{NA} \\
\mathsf{sameThread}(a, b) &\stackrel{\text{def}}{=} \mathsf{tid}(a) = \mathsf{tid}(b) \\
\mathsf{isrmw}(a) &\stackrel{\text{def}}{=} \mathsf{isread}(a) \wedge \mathsf{iswrite}(a) \\
\mathsf{isSC}(a) &\stackrel{\text{def}}{=} \mathsf{mode}(a) = \text{SC} \\
\mathsf{rsElem}(a, b) &\stackrel{\text{def}}{=} \mathsf{sameThread}(a, b) \vee \mathsf{isrmw}(b) \\
\mathsf{isAcq}(a) &\stackrel{\text{def}}{=} \mathsf{mode}(a) \sqsupseteq \text{ACQ} \\
\mathsf{isRel}(a) &\stackrel{\text{def}}{=} \mathsf{mode}(a) \sqsupseteq \text{REL} \\
\mathsf{rseq}(a, b) &\stackrel{\text{def}}{=} a = b \vee \mathsf{rsElem}(a, b) \wedge mo(a, b) \wedge (\forall c.\ mo(a, c) \wedge mo(c, b) \Rightarrow \mathsf{rsElem}(a, c)) \\
\mathsf{sw}(a, b) &\stackrel{\text{def}}{=} \exists c, d.\ \begin{array}{l} \neg\mathsf{sameThread}(a, b) \wedge \mathsf{isRel}(a) \wedge \mathsf{isAcq}(b) \wedge \mathsf{rseq}(c, rf(d)) \\ \wedge (a = c \vee \mathsf{isfence}(a) \wedge sb^+(a, c)) \wedge (d = b \vee \mathsf{isfence}(d) \wedge sb^+(d, b)) \end{array} \\
\mathsf{hb} &\stackrel{\text{def}}{=} (sb \cup \mathsf{sw} \cup asw)^+ \\
\mathsf{Racy} &\stackrel{\text{def}}{=} \exists a, b.\ \begin{array}{l} \mathsf{isaccess}(a) \wedge \mathsf{isaccess}(b) \wedge \mathsf{loc}(a) = \mathsf{loc}(b) \wedge a \neq b \\ \wedge (\mathsf{iswrite}(a) \vee \mathsf{iswrite}(b)) \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \wedge \neg(\mathsf{hb}(a, b) \vee \mathsf{hb}(b, a)) \end{array} \\
\mathsf{Observation} &\stackrel{\text{def}}{=} \{(a, b) \mid mo(a, b) \wedge \mathsf{loc}(a) = \mathsf{loc}(b) = \texttt{world}\}
\end{aligned}
$$

Figure 2.12: Auxiliary definitions for a C11 execution $(lab, sb, asw, rf, mo, sc)$.

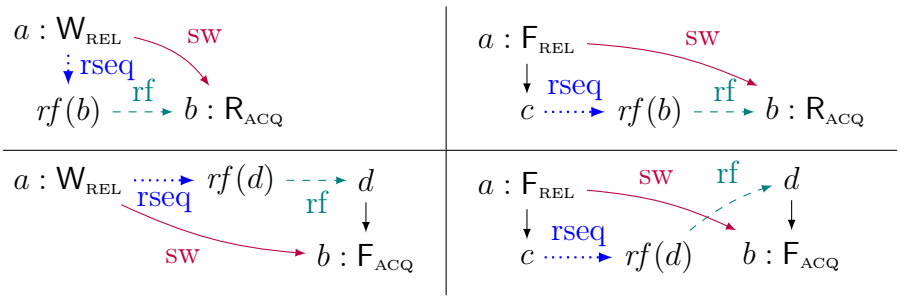


Figure 2.13: Illustration of the "synchronizes-with" definition: the four cases inducing an sw edge.

- *Synchronizes-with* (sw) relates each release write with the acquire reads that read from some write in its release sequence (rseq). This sequence includes the release write and certain subsequent writes in modification order that belong to the same thread or are RMW operations. The sw relation also relates fences under similar conditions. Roughly speaking, a release fence turns succeeding writes in *sb* into releases and an acquire fence turns preceding reads into acquires. (For details, see the definition in Figure 2.12 and the illustration in Figure 2.13.)
- *Happens-before* (hb) is a partial order on actions formalising the intuition that one action was completed before the other. In the C11 subset we consider, $\mathsf{hb} = (sb \cup \mathsf{sw} \cup asw)^+$.

We refer to a pair of an opsem and a witness (O, W) as a *candidate execution*. A candidate execution is said to be *consistent* if it satisfies the axioms of the memory model, which will be presented shortly. The model finally checks if none of the consistent executions contains an *undefined behaviour*, arising from a *race* (two conflicting accesses not related by hb)[4] or a *memory error* (accessing an unallocated location), where two accesses are conflicting if they are to the same address, at least one is a write, and at least one is non-atomic. Programs that exhibit an undefined behaviour in one of their consistent executions are undefined; programs that do not exhibit any undefined behaviour are called *well defined*, and their semantics is given by the set of their consistent executions.

Consistent Executions. According to the C11 model, a candidate execution $(lab, sb, asw, rf, mo, sc)$ is consistent if all of the properties shown in Figure 2.14 hold.

(ConsSB) Sequenced-before relates only same-thread actions.

(ConsMO) Writes on the same location are totally ordered by *mo*.

(ConsSC) The *sc* relation must be a total order over SC actions and include both hb and *mo* restricted to SC actions. This in effect means that SC actions are globally synchronised.

(ConsRFdom) The reads-from map, *rf*, is defined for exactly those read actions for which the execution contains an earlier write to the same location.

(ConsRF) Each entry in the reads-from map, *rf*, should map a read to a write to the same location and with the same value.

(ConsRFna) If a read reads from a write and either the read or the write are non-atomic, then the write must have happened before the read. Batty et al. [BOS+11] additionally require the write to be *visible*: i.e. not to have been overwritten by another write that happened before the read. This extra condition is unnecessary, as it follows from (CohWR).

(SCReads) SC reads are restricted to read only from the immediately preceding SC write to the same location in *sc* order or from a non-SC write that has not happened before that immediately preceding SC write.

(IrrHB) The happens-before order, hb, must be irreflexive: an action cannot happen before itself.

[4] The standard distinguishes between races arising from accesses of different threads, which it calls *data races*, and from those of the same thread, which it calls *unsequenced races*. The standard says unsequenced races can occur even between atomic accesses.

$$\forall a, b.\ \ sb(a,b) \implies \mathsf{tid}(a) = \mathsf{tid}(b) \qquad \text{(ConsSB)}$$

$$\mathsf{order}(\mathsf{iswrite}, mo) \wedge \forall \ell.\ \ \mathsf{total}(\mathsf{iswrite}_\ell, mo) \qquad \text{(ConsMO)}$$

$$\mathsf{order}(\mathsf{isSC}, sc) \wedge \mathsf{total}(\mathsf{isSC}, sc) \wedge (\mathsf{hb} \cup mo) \cap (\mathsf{isSC} \times \mathsf{isSC}) \subseteq sc \qquad \text{(ConsSC)}$$

$$\forall b.\ \ (\exists c.\ rf(b) = c) \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a,b) \qquad \text{(ConsRFdom)}$$

$$\forall a, b.\ \ rf(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \qquad \text{(ConsRF)}$$

$$\forall a, b.\ \ rf(b) = a \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \implies \mathsf{hb}(a,b) \qquad \text{(ConsRFna)}$$

$$\forall a, b.\ \ rf(b) = a \wedge \mathsf{isSC}(b) \implies \mathsf{imm}(\mathsf{scr}, a, b) \vee (\neg\mathsf{isSC}(a) \wedge \nexists x.\ \mathsf{hb}(a,x) \wedge \mathsf{imm}(\mathsf{scr}, x, b)) \qquad \text{(SCReads)}$$

$$\nexists a.\ \ \mathsf{hb}(a,a) \qquad \text{(IrrHB)}$$

$$\nexists a, b.\ \ rf(b) = a \wedge \mathsf{hb}(b,a) \qquad \text{(ConsRFhb)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a,b) \wedge mo(b,a) \qquad \text{(CohWW)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a,b) \wedge mo(rf(b), rf(a)) \qquad \text{(CohRR)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a,b) \wedge mo(rf(b), a) \qquad \text{(CohWR)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a,b) \wedge mo(b, rf(a)) \qquad \text{(CohRW)}$$

$$\forall a, b.\ \ \mathsf{isrmw}(a) \wedge rf(a) = b \implies \mathsf{imm}(mo, b, a) \qquad \text{(AtRMW)}$$

$$\forall a, b, \ell.\ \ lab(a) = lab(b) = \mathsf{A}(\ell) \implies a = b \qquad \text{(ConsAlloc)}$$

$$\forall a, b, x.\ \ (\mathsf{isfence}(x) \wedge \mathsf{isread}(b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a) \wedge \mathsf{imm}(sc, a, x) \wedge sb^+(x,b)) \implies (a = rf(b) \vee mo(a, rf(b))) \qquad \text{(SCFences1)}$$

$$\forall a, b, x.\ \ (\mathsf{isfence}(x) \wedge \mathsf{isread}(b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a) \wedge sb^+(a,x) \wedge sc(x,b)) \implies (a = rf(b) \vee mo(a, rf(b))) \qquad \text{(SCFences2)}$$

$$\forall a, b, x, y.\ \ (\mathsf{isfence}(x) \wedge \mathsf{isfence}(y) \wedge \mathsf{isread}(b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a) \wedge sc(x,y) \wedge sb^+(a,x) \wedge sb^+(y,b)) \implies (a = rf(b) \vee mo(a, rf(b))) \qquad \text{(SCFences3)}$$

$$\forall a, b, x, y.\ \ (\mathsf{isfence}(x) \wedge \mathsf{isfence}(y) \wedge \mathsf{iswrite}(b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a) \wedge sc(x,y) \wedge sb^+(a,x) \wedge sb^+(y,b)) \implies mo(a,b) \qquad \text{(SCFences4)}$$

$$\forall a, b, y.\ \ (\mathsf{isfence}(y) \wedge \mathsf{iswrite}(b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a) \wedge sc(a,y) \wedge \wedge sb^+(y,b)) \implies mo(a,b) \qquad \text{(SCFences5)}$$

$$\forall a, b, x.\ \ (\mathsf{isfence}(x) \wedge \mathsf{iswrite}(b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a) \wedge sc(x,b) \wedge sb^+(a,x) \wedge \implies mo(a,b) \qquad \text{(SCFences6)}$$

where
$$\mathsf{order}(P,R) \stackrel{\mathsf{def}}{=} (\nexists a.\ R(a,a)) \ \wedge\ (R^+ \subseteq R) \ \wedge\ (R \subseteq P \times P)$$
$$\mathsf{imm}(R,a,b) \stackrel{\mathsf{def}}{=} R(a,b) \wedge \nexists c.\ R(a,c) \wedge R(c,b)$$
$$\mathsf{total}(P,R) \stackrel{\mathsf{def}}{=} (\forall a, b.\ P(a) \wedge P(b) \implies a = b \vee R(a,b) \vee R(b,a))$$
$$\mathsf{scr}(a,b) \stackrel{\mathsf{def}}{=} sc(a,b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a)$$

Figure 2.14: Axioms satisfied by consistent C11 executions, $\mathsf{Consistent}(lab, sb, asw, rf, mo, sc)$.

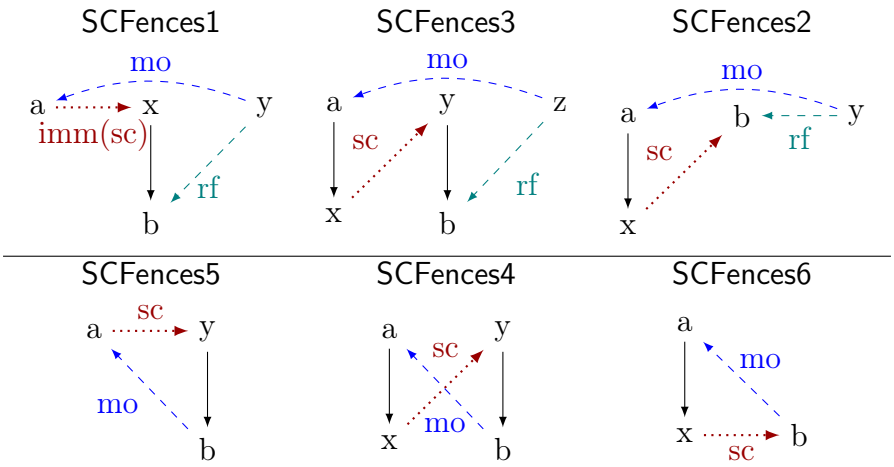


Figure 2.15: Patterns forbidden by the SCFences axioms.

(ConsRFhb) A read cannot read from a future write.

(CohWW, CohRR, CohWR, CohRW) Next, we have four coherence properties relating *mo*, hb, and *rf* on accesses to the same location. These properties require that *mo* never contradicts hb or the observed read order, and that *rf* never reads values that have been overwritten by more recent actions that happened before the read.

(AtRMW) Read-modify-write accesses execute atomically: they read from the immediately preceding write in *mo*.

(ConsAlloc) The same location cannot be allocated twice by different allocation actions. (This axiom is sound because for simplicity we do not model deallocation. The C11 model by Batty et al. [BOS+11] does not even model allocation.)

(SCFences1, SCFences2, SCFences3, SCFences4, SCFences5, SCFences6) These six properties explain in detail how SC fences restrict *mo* for writes nearby. More precisely they forbid the patterns shown in Figure 2.15.

Chapter 3

Issues with the C11 model and possible improvements

In this chapter we will list unintuitive consequences of the axioms of the C11 model, and propose improvements to the model where possible.

We will first explain some known difficulties related to consume accesses and SC fences. They led us to ignore these parts of the language in the rest of our treatment of C11.

3.0.1 Consume accesses

The formal C11 model we saw in Section 2.3.3 did not include CONSUME. Remember, a CONSUME read is supposed to order the store it reads from (provided it is at least release) with any read that has a data or address dependency on it. The problem is how to define such dependencies. In the standard, it is done syntactically. As a result there is a dependency between the two loads in the following code:

```
r = x.load(consume);
y = *(z+r-r);
```

but there is naturally no dependency between them in the following:

```
r = x.load(consume);
y = *z;
```

Any optimising compiler will transform the first fragment into the second one. Since there is no longer a dependency, the compiler must now introduce a fence before the second read on ARM and Power to prevent the processor from reordering the two reads. But knowing this would require the compiler to track every dependency in the program throughout the compilation pipeline. In particular, as dependencies are not necessarily local, it would break modular compilation. As a consequence, compilers just insert a fence every time on ARM/Power, treating consume as the (less efficient in theory) acquire. Several proposals have been made to improve this situation [McK16] and are under discussion by the C++ standards committee.

As the presence of CONSUME significantly complicates the model (for example it makes hb not transitive), and it is not usefully implementable, we will ignore it in the rest of this thesis.

3.0.2 SC fences

The semantics of SC fences is also problematic, as despite their name they cannot restore full sequential consistency. Consider for example the IRIW example (Figure 2.3). If all the accesses are SC, then only the sequentially consistent outcomes can happen. But if the accesses are all RLX, no amount of SC fences can prevent the non-sequentially consistent outcome. This can be seen by a look at the axioms specific to SC fences:

- SCFences1, SCFences5, SCFences6 do not apply as they require a SC write;
- SCFences2 does not apply as it requires a SC read;
- SCFences3 only means that any SC fence sb after the writes must also be sc after any fence between the reads;
- SCFences4 does not apply as it requires two writes to the same location, one with a SC fence after it and the other with one before it.

This is unlikely to have been intended by the standards authors, as SC fences are supposed to be the strongest form of synchronisation, and their implementation on all platforms actually prevent the IRIW anomalous behaviour. For this reason, and because of their complexity, we have also decided to ignore SC fences in our treatment of C11.

3.1 Sanity properties

Now that we have identified a useful subset of C11, it is natural to check whether it has the properties we expect.

3.1.1 Weakness of the SCReads axiom

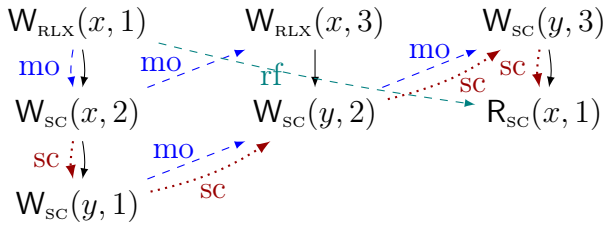
One desirable property of a memory model is that adding synchronisation to a program introduces no new behaviour (other than deadlock). We have seen that RLX is supposed to be the weakest memory attribute, and SC the strongest. So it should always be sound to replace a RLX access by a SC one. Unfortunately, it is not correct in C11 as it stands. Consider the example in Figure 3.1.

Coherence of the relaxed loads in the final thread forces the mo-orderings to be as shown in the execution on the bottom of the figure. Now, the question is whether the SC-load can read from the first store to x and return $r = 1$. In the program as shown, it cannot, because that store happens before the $x.\mathsf{store}(2, \text{SC})$ store, which is the immediate sc-preceding store to x before the load. If, however, we also make the $x.\mathsf{store}(3, \text{RLX})$ be sequentially consistent, then it becomes the immediately sc-preceding store to x, and hence reading $r = 1$ is no longer blocked. The SCReads axiom places an odd restriction on where a sequentially consistent read can read from. The problem arises from the case where the source of the read is a non SC write. In this case, the axiom forbids that write to happen before the immediately sc-preceding write to the same location. It may, however, happen before an earlier write in the sc order.

We propose to strengthen the SCReads axiom by requiring there not to be a happens before edge between $rf(b)$ and any same-location write sc-prior to the read, as follows:

$$\forall a, b.\ \ rf(b) = a \wedge \mathsf{isSC}(b) \implies \mathsf{imm}(\mathsf{scr}, a, b) \vee (\neg\mathsf{isSC}(a) \wedge \nexists x.\ \mathsf{hb}(a, x) \wedge \mathsf{scr}(x, b)) \qquad (\mathsf{SCReads}')$$

$$
\begin{array}{l}
x.\mathsf{store}(1, \text{RLX}); \\
x.\mathsf{store}(2, \text{SC}); \\
y.\mathsf{store}(1, \text{SC});
\end{array}
\;\Big\|\;
\begin{array}{l}
x.\mathsf{store}(3, \text{RLX}); \\
y.\mathsf{store}(2, \text{SC});
\end{array}
\;\Big\|\;
\begin{array}{l}
y.\mathsf{store}(3, \text{SC}); \\
r = x.\mathsf{load}(\text{SC});
\end{array}
\;\Big\|\;
\begin{array}{l}
s_1 = x.\mathsf{load}(\text{RLX}); \\
s_2 = x.\mathsf{load}(\text{RLX}); \\
s_3 = x.\mathsf{load}(\text{RLX}); \\
t_1 = y.\mathsf{load}(\text{RLX}); \\
t_2 = y.\mathsf{load}(\text{RLX}); \\
t_3 = y.\mathsf{load}(\text{RLX});
\end{array}
$$



Behaviour in question: $r = s_1 = t_1 = 1 \;\wedge\; s_2 = t_2 = 2 \;\wedge\; s_3 = t_3 = 3$

Figure 3.1: A weird consequence of the SCReads axiom: strengthening the $x.\mathsf{store}(3, \text{RLX})$ into $x.\mathsf{store}(3, \text{SC})$ introduces new behaviour.

instead of the original:

$$\forall a, b.\quad rf(b) = a \wedge \mathsf{isSC}(b) \implies \mathsf{imm}(\mathsf{scr}, a, b) \vee (\neg\mathsf{isSC}(a) \wedge \nexists x.\ \mathsf{hb}(a, x) \wedge \mathsf{imm}(\mathsf{scr}, x, b)) \tag{SCReads}$$

Going back to the program in Figure 3.1, this stronger axiom rules out reading $r = 1$, a guarantee that is provided by the suggested compilations of C11 atomic accesses to x86/Power/ARM.

Batty et al. [BDW16] later noticed that this modified axiom enables more efficient simulation of C11 programs, and proposed to further simplify the treatment of SC atomics, treating all SC atomics (including the fences) in one single axiom.

3.1.2 Relaxed atomics allow causality cycles

The C11 semantics for relaxed atomics surprisingly allow causality cycles. To understand why, consider the following programs (with `x` and `y` initialized at 0):

$$
\begin{array}{l}
r_1 = x.\mathsf{load}(\text{RLX}); \\
y.\mathsf{store}(1, \text{RLX});
\end{array}
\;\Big\|\;
\begin{array}{l}
r_2 = y.\mathsf{load}(\text{RLX}); \\
x.\mathsf{store}(1, \text{RLX});
\end{array}
\tag{LB}
$$

$$
\begin{array}{l}
\mathsf{if}\ (x.\mathsf{load}(\text{RLX})) \\
\quad y.\mathsf{store}(1, \text{RLX});
\end{array}
\;\Big\|\;
\begin{array}{l}
\mathsf{if}\ (y.\mathsf{load}(\text{RLX})) \\
\quad x.\mathsf{store}(1, \text{RLX});
\end{array}
\tag{CYC}
$$

By the rules that we introduced in Section 2.3.3, it is clear that the LB program can end with $x = y = 1$. This is to be expected, since relaxed atomics are intended to be compilable to all mainstream architectures without introducing fences, and this behaviour is allowed on Power and ARM.

Sadly, the opsem with that result is also an opsem of CYC, and this program is well defined (since every access is atomic). Since the C11 model checks the consistency of executions opsem by opsem, it must by necessity also allow $x = y = 1$ for that second program. This result is highly counter-intuitive as it does not happen on any platform,

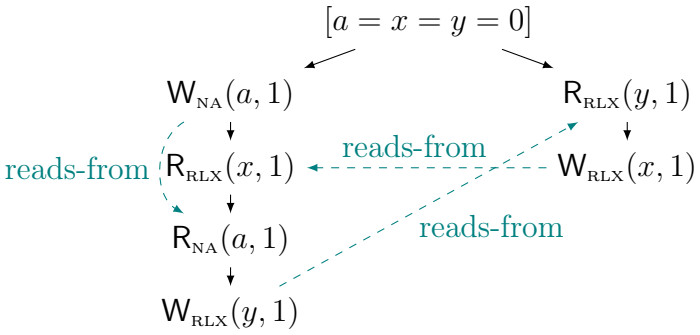


Figure 3.2: Execution resulting in $a = x = y = 1$.

and would require a speculation of the write in one of the threads, observable by the other thread. We call these kind of situations where a write needs to be observed by another thread in order to have a chance to happen a *causality cycle*.

Several authors have observed that causality cycles make code verification infeasible [BDG13, ND13, VN13]. We show that the situation is even worse than that, because we can exploit them to show that standard program transformations are unsound. Consider:

$$a = 1; \;\Big\|\; \begin{array}{l}\mathsf{if}\ (x.\mathsf{load}(\text{RLX}))\\ \quad \mathsf{if}\ (a)\\ \quad\quad y.\mathsf{store}(1, \text{RLX});\end{array} \;\Big\|\; \begin{array}{l}\mathsf{if}\ (y.\mathsf{load}(\text{RLX}))\\ \quad x.\mathsf{store}(1, \text{RLX});\end{array} \qquad \text{(SEQ)}$$

First, notice that there is no consistent execution in which the load of a occurs. We show this by contradiction. Suppose that there is an execution in which a load of a occurs. In such an execution the load of a can only return 0 (the initial value of a) because the store $a = 1$ does not happen before it (because it is in a different thread that has not been synchronised with) and non atomic loads must return the latest write that happens before them(ConsRFna). Therefore, in this execution the store to y does not happen, which in turn means that the load of y cannot return 1 and the store to x also does not happen. Then, x cannot read 1, and thus the load of a does not occur. As a consequence this program is *not racy*: since the load of a does not occur in any execution, there are no executions with conflicting accesses on the same non atomic variable. We conclude that the only possible final state is $a = 1 \wedge x = y = 0$.

Now, imagine we apply sequentialisation, collapsing the first two threads and moving the assignment to the start:

$$\begin{array}{l}a = 1;\\ \mathsf{if}\ (x.\mathsf{load}(\text{RLX}))\\ \quad \mathsf{if}\ (a)\\ \quad\quad y.\mathsf{store}(1, \text{RLX});\end{array} \;\Big\|\; \begin{array}{l}\mathsf{if}\ (y.\mathsf{load}(\text{RLX}))\\ \quad x.\mathsf{store}(1, \text{RLX});\end{array}$$

Running the resulting code can lead to an execution, formally depicted in Figure 3.2, in which the load of a actually returns the value 1 since the store to a now happens before (via program-order) the load. This results in the final state $a = x = y = 1$, which is not possible for the initial program.

Expression Linearisation is Unsound. A simple variation of sequentialisation is expression evaluation order *linearisation*, a transformation that adds an *sb* arrow between

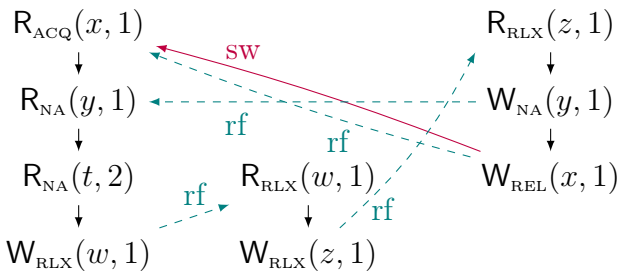


Figure 3.3: Execution generating new behaviour if the expression evaluation order is linearised.

two actions of the same thread and that every compiler is bound to perform. This transformation is unsound as demonstrated below:

$$\begin{array}{l} t = x.\mathsf{load}(\text{ACQ})+y; \\ \mathsf{if}\ (t == 2) \\ \quad w.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (w.\mathsf{load}(\text{RLX})) \\ \quad z.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (z.\mathsf{load}(\text{RLX})) \\ \quad y = 1; \\ \quad x.\mathsf{store}(1, \text{REL}); \end{array}$$

The only possible final state for this program has all variables, including t, set to zero. Indeed, the store $y = 1;$ does not happen before the load of y, which can then return only 0. However, if the $t = x.\mathsf{load}(\text{ACQ}) + y;$ is linearised into $t = x.\mathsf{load}(\text{ACQ}); t = t + y;$, then a synchronisation on x induces an order on the accesses to y, and the execution shown in Figure 3.3 is allowed.

Such a deeply counter-intuitive result effectively makes it impossible to use the C11 model as the basis for a compiler intermediate representation, such as LLVM IR. It also makes it extremely hard to make tools that can reason about arbitrary C11 programs.

Strengthening is Unsound. There are other transformations that would be intuitive but fail on similar counterexamples. For example, even after fixing the problem with SC semantics, strengthening a memory order is still unsound. The following example shows in particular that replacing a relaxed atomic store with a release atomic store is unsound in C11. Consider

$$\begin{array}{l} a = 1; \\ z.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{RLX})) \\ \quad \mathsf{if}\ (z.\mathsf{load}(\text{ACQ})) \\ \quad\quad \mathsf{if}\ (a) \\ \quad\quad\quad y.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (y.\mathsf{load}(\text{RLX})) \\ \quad x.\mathsf{store}(1, \text{RLX}); \end{array}$$

As in the SEQ program, the load of a cannot return 1 because the store to a does not happen before it. Therefore, the only final state is $a = z = 1 \wedge x = y = 0$. If, however, we make the store of z a release store, then it synchronises with the acquire load, and it is easy to build a consistent execution with final state $a = z = x = y = 1$. A symmetric counterexample can be constructed for strengthening a relaxed load to an acquire load.

Roach Motel Reorderings are Unsound. Roach motel reorderings are a class of optimizations that let compilers move accesses to memory into synchronised blocks, but not to move them out: the intuition is that it is always safe to move more computations (including memory accesses) inside critical sections. In the context of C11, roach motel reorderings would allow moving non atomic accesses after an acquire read (which behaves as a lock operation) or before a release write (which behaves as an unlock operation).

However the following example program shows that in C11 it is unsound to move a non atomic store before a release store.

$$
\begin{array}{l||l||l}
z.\mathsf{store}(1, \text{REL}); & \mathsf{if}\ (x.\mathsf{load}(\text{RLX})) & \\
a = 1; & \quad \mathsf{if}\ (z.\mathsf{load}(\text{ACQ})) & \mathsf{if}\ (y.\mathsf{load}(\text{RLX})) \\
& \quad\quad \mathsf{if}\ (a) & \quad x.\mathsf{store}(1, \text{RLX}); \\
& \quad\quad\quad y.\mathsf{store}(1, \text{RLX}); &
\end{array}
$$

As before, the only possible final state of this program is $a = z = 1$ and $x = y = 0$. If, however, we reorder the two stores in the first thread, we get a consistent execution leading to the final state $a = z = x = y = 1$. Again, we can construct a similar example showing that reordering over an acquire load is also not allowed by C11.

3.1.2.1 Proposed fixes

A first, rather naive solution is to permit causality cycles, but drop the offending ConsRFna axiom. As we will show in Sections 4.2.2 and 4.2.4, this solution allows all the optimizations that were intended to be sound on C11. It is, however, of dubious usefulness as it gives extremely weak guarantees to programmers.

The DRF theorem—stating that programs whose sequential consistent executions have no data races, have no additional relaxed behaviours besides the SC ones—does not hold. As a counterexample, take the CYC program from before, replacing the relaxed accesses by non atomic ones.

Arf: Forbidding $(\mathsf{hb} \cup rf)$ Cycles A second, much more reasonable solution is to try to rule out causality cycles. Ruling out causality cycles, while allowing non causal loops in $\mathsf{hb} \cup rf$ is, however, difficult and cannot be done by stating additional axioms over single executions. This is essentially because the offending execution of the CYC program is also an execution of the LB program.

As an approximation, we can rule out all $(\mathsf{hb} \cup rf)$ cycles, by stating the following axiom:

$$\mathsf{acyclic}(\mathsf{hb} \cup \{(a, b) \mid rf(b) = a\}) \tag{Arf}$$

This solution has been proposed before by Boehm and Demsky [BD14] and also by Vafeiadis and Narayan [VN13]. Here, however, we take a subtly different approach from the aforementioned proposals in that besides adding the Arf axiom, we also drop the problematic ConsRFna axiom.

In Sections 4.2.2 and 4.2.4 we show that this model allows the same optimizations as the naive one (i.e., all the intended ones), except the reordering of atomic reads over atomic writes.

It is however known to make relaxed accesses more costly on ARM/Power, as there must be either a bogus branch or a lightweight fence between every shared load and shared store [BD14].

Arfna: Forbidding Only Non-Atomic Cycles Another approach is to instead make more behaviours consistent, so that the non atomic accesses in the SEQ example from the introduction can actually occur and race. The simplest way to do this is to replace ConsRFna by

$$\mathsf{acyclic}(\mathsf{hb} \cup \{(rf(b), b) \mid \mathsf{isNA}(b) \vee \mathsf{isNA}(rf(b))\}) \tag{Arfna}$$

A non atomic load can read from a concurrent write, as long as it does not cause a causality cycle.

This new model has several nice properties. First, it is weaker than C11 in that it allows all behaviours permitted by C11. This entails that any compilation strategy proved correct from C11 to hardware memory models, such as to x86-TSO and Power, remains correct in the modified model (contrary to the previous fix).

Theorem 1. *If* $\mathsf{Consistent}_{\mathrm{C11}}(X)$*, then* $\mathsf{Consistent}_{\mathsf{Arfna}}(X)$*.*

Proof. Straightforward, since by the ConsRFna condition,

$$\{(rf(b), b) \mid \mathsf{isNA}(b) \vee \mathsf{isNA}(rf(b))\} \subseteq \mathsf{hb}$$

and hence Arfna follows from IrrHB. □

Second, this model is not much weaker than C11. More precisely, it only allows more racy behaviours.

Theorem 2. *If* $\mathsf{Consistent}_{\mathsf{Arfna}}(X)$ *and not* $\mathsf{Racy}(X)$*, then* $\mathsf{Consistent}_{\mathsf{C11}}(X)$*.*

Note that the definition of racy executions, $\mathsf{Racy}(X)$, does not depend on the axioms of the model, and is thus the same for all memory models considered here.

Finally, it is possible to reason about this model as most reasoning techniques on C11 remain true. In particular, in the absence of relaxed accesses, this model is equivalent to the Arf model. We are thus able to use the program logics that have been developed for C11 (namely, RSL [VN13] and GPS [TVD14]) to also reason about programs in the Arfna model.

However, we found that reordering non atomic loads past non atomic stores is forbidden in this model, as shown by the following example:

if $(x.\mathsf{load}(\mathrm{RLX}))$ {	if $(y.\mathsf{load}(\mathrm{RLX}))$
$t = a;$	if (b) {
$b = 1;$	$a = 1;$
if (t) $y.\mathsf{store}(1, \mathrm{RLX});$	$x.\mathsf{store}(1, \mathrm{RLX});$
}	}

In this program, the causality cycle does not occur, because for it to happen, an $(\mathsf{hb} \cup rf)$-cycle must also occur between the a and b accesses (and that is ruled out by our axiom). However, if we swap the non atomic load of a and store of b in the first thread, then the causality cycle becomes possible, and the program is racy. Introducing a race is clearly unsound, so compilers are not allowed to do such reorderings (note that these accesses are non atomic and adjacent). It is not clear whether such a constraint would be acceptable in C/C++ compilers.

3.2 Further simplifications and strengthenings of the model

3.2.1 Simplifying the coherency axioms

Since *mo* is a total order on writes to the same location, and hb is irreflexive, the CohWW axiom is actually equivalent to the following one:

$$\forall a, b, \ell.\ \mathsf{hb}(a, b) \wedge \mathsf{iswrite}_{\ell}(a) \wedge \mathsf{iswrite}_{\ell}(b) \implies mo(a, b) \qquad (\mathsf{ConsMOhb})$$

The equivalence can be derived by a case analysis on how mo orders a and b. (For what it is worth, the C/C++ standards as well as the formal model of Batty et al. [BOS+11] include both axioms even though, as we show, one of them is redundant.)

Next, we show that the coherence axioms can be restated in terms of a single acyclicity axiom. To state this axiom, we need some auxiliary definitions. We say that a read, a, *from-reads* a different write, b, denoted $\mathsf{fr}(a, b)$, if and only if $a \neq b$ and $mo(rf(a), b)$. (Note that we need the $a \neq b$ condition because RMW actions are simultaneously both reads and writes.) We define the communication order, com, as the union of the modification order, the reads-from map, and the from-reads relation.

$$\begin{aligned} \mathsf{fr}(a,b) &\stackrel{\text{def}}{=} mo(rf(a), b) \wedge a \neq b \\ \mathsf{com}(a,b) &\stackrel{\text{def}}{=} mo(a,b) \vee rf(b) = a \vee \mathsf{fr}(a,b) \end{aligned}$$

In essence, for every location ℓ, com^+ relates the writes to and initialised reads from that location, ℓ. Except for uninitialised reads and loads reading from the same write, com^+ is a total order on all accesses of a given location, ℓ.

We observe that all the violations of the coherence axioms are cyclic in $\{(a, b) \in \mathsf{hb} \mid \mathsf{loc}(a) = \mathsf{loc}(b)\} \cup \mathsf{com}$ (see Figure 3.4). This is not accidental: from Shasha and Snir [SS88a] we know that any execution acyclic in $\mathsf{hb} \cup \mathsf{com}$ is sequentially consistent, and coherence essentially guarantees sequential consistency on a per-location basis. We can also observe that the same rule is used in the "Herding Cats" model (see Section 2.2.2). Doug Lea mentionned that a similar characterisation can be found in the appendix of [LV15].

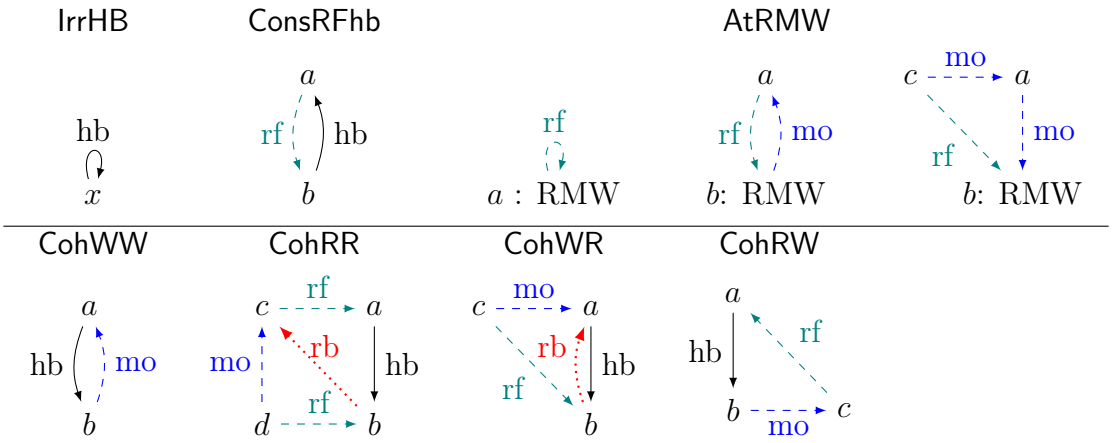


Figure 3.4: Executions violating the coherence axioms: all contain a cycle in $\{(a, b) \in \mathsf{hb} \mid \mathsf{loc}(a) = \mathsf{loc}(b)\} \cup \mathsf{com}$.

Based on this observation, we consider the following axiom stating that the union of hb restricted to relate same-location actions and com is acyclic.

$$\mathsf{acyclic}(\{(a, b) \in \mathsf{hb} \mid \mathsf{loc}(a) = \mathsf{loc}(b)\} \cup \mathsf{com}) \qquad (\mathsf{Coh})$$

This axiom is equivalent to the conjunction of seven C11 axioms as shown in the following theorem:

Theorem 3. *Assuming* ConsMO *and* ConsRF *hold, then*

$$\mathsf{Coh} \iff \begin{pmatrix} \mathsf{IrrHB} \wedge \mathsf{ConsRFhb} \wedge \mathsf{CohWW} \wedge \\ \mathsf{CohRR} \wedge \mathsf{CohWR} \wedge \mathsf{CohRW} \wedge \mathsf{AtRMW} \end{pmatrix}.$$

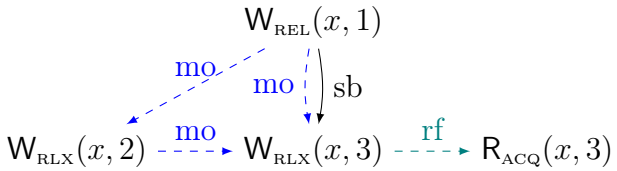
Proof (sketch). In the ($\Rightarrow$) direction, it is easy to see that all the coherence axiom violations exhibit cycles (see Fig. 3.4). In the other direction, careful analysis reveals that these are the only possible cycles—any larger ones can be shortened as *mo* is a total order. □

3.2.2 The definition of release sequences

The definition of release sequences in the C11 model is too weak, as shown by the following example.

$$x.\mathsf{store}(2, \text{RLX}); \;\Big\|\; \begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \\ x.\mathsf{store}(3, \text{RLX}); \end{array} \;\Big\|\; \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ}) == 3) \\ \quad \mathsf{print}(y); \end{array}$$

In this program, assuming the test condition holds, the acquire load of x need not synchronise with the release store even though it reads from a store that is sequenced after the release, and hence the program is racy. The reason is that the seemingly irrelevant store of $x.\mathsf{store}(2, \text{RLX})$ can interrupt the release sequence as shown in the following execution snippet.



In the absence, however, of the first thread, the acquire and the release do synchronise and the program is well defined.

As a fix for the release sequences definition, we propose to replace the definition of release sequences by the least fixed point of the following recursive definition (with respect to $\subseteq$) :

$$\mathsf{rseq}_{\mathsf{RSnew}}(a, b) \stackrel{\mathsf{def}}{=} a = b \vee (\mathsf{sameThread}(a, b) \wedge mo(a, b)) \vee (\mathsf{isrmw}(b) \wedge \mathsf{rseq}_{\mathsf{RSnew}}(a, rf(b)))$$

Our release sequences are not defined in terms of *mo* sequences, but rather in terms of *rf* sequences. Either b should belong to the same thread as a, or there should be a chain of RMW actions reading from one another connecting b to a write in the same thread as a.

In the absence of uninitialised RMW accesses, this change strengthens the semantics. Every consistent execution in the revised model is also consistent in the original model. Despite being a strengthening, it does not affect the compilation results to x86, Power, and ARM. The reason is that release sequences do not play any role on x86, while on Power and ARM the compilation of release writes and fences issues a memory barrier that affects all later writes of the same thread, not just an uninterrupted *mo*-sequence of such writes.

3.2.3 Intra-thread synchronisation is needlessly banned

A final change is to remove the slightly odd restriction that actions from the same thread cannot synchronise.[1] This change allows us to give meaning to more programs. In the original model, the following program has undefined behaviour:

$$\begin{array}{l} \#\mathsf{define}\ f(x, y)\ (x.\mathsf{CAS}(1, 0, \text{ACQ})?(y{+}{+}, x.\mathsf{store}(1, \text{REL})) : 0) \\ f(x, y) + f(x, y) \end{array}$$

[1] This restriction breaks monotonicity in the presence of consume reads.

That is, although f uses x as a lock to protect the increments of y, and therefore the y accesses could never be adjacent in an interleaving semantics, the model does not treat the x-accesses as synchronising because they belong to the same thread. Thus, the two increments of y are deemed to race with one another.

As we believe that this behaviour is highly suspicious, we have also considered an adaptation of the C11 model, where we set

$$\mathsf{sameThread}_{\mathsf{STnew}}(a, b) \stackrel{\mathsf{def}}{=} sb^{+}(a, b)$$

rather than $\mathsf{tid}(a) = \mathsf{tid}(b)$. We have proved that with the new definition, we can drop the $\neg\mathsf{sameThread}(a, b)$ conjunct from the sw definition without affecting hb.

Since, by the ConsSB axiom, every sb edge has the same thread identifiers, the change also strengthens the model by assigning defined behaviour to more programs.

Chapter 4

Correct optimizations in the C11 model

A great number of compiler optimizations have been proven correct over the years in the literature, at different levels of abstraction (from source code to assembly). However, most of these proofs were done assuming a sequential model of execution, or at least sequential consistency.

Ševčík showed that a large class of elimination and reordering transformations are correct (that is, do not introduce any new behaviour when the optimized code is put in an arbitrary data-race free context) in an idealised DRF model [Sev08, Sev11].

In this chapter we adapt and extend his result to the C11/C++11 model (Section 2.3.3), finding under which conditions compiler optimizations that are sound in the sequential setting remain so for C11 compilers.

Summary of the Models to be Considered As the four problems explained in Chapter 3 are independent and we have proposed fixes to each problem, we can look at the product of the fixes:

$$\overbrace{\begin{pmatrix}\mathsf{ConsRFna}\\ \mathsf{Naive}\\ \mathsf{Arfna}\\ \mathsf{Arf}\end{pmatrix}}^{\S 3.1.2} \times \overbrace{\begin{pmatrix}\mathsf{SCorig}\\ \mathsf{SCnew}\end{pmatrix}}^{\S 3.1.1} \times \overbrace{\begin{pmatrix}\mathsf{RSorig}\\ \mathsf{RSnew}\end{pmatrix}}^{\S 3.2.2} \times \overbrace{\begin{pmatrix}\mathsf{STorig}\\ \mathsf{STnew}\end{pmatrix}}^{\S 3.2.3}$$

We use tuple notation to refer to the individual models. For example, we write $(\mathsf{ConsRFna}, \mathsf{SCorig}, \mathsf{RSorig}, \mathsf{STorig})$ for the model corresponding to the 2011 C and C++ standards.

In a first time, we will focus on the optimisations that we have observed GCC and Clang do (see Chapter 5 for details), and prove their correctness in the model currently defined by the standards: $(\mathsf{ConsRFna}, \mathsf{SCorig}, \mathsf{RSorig}, \mathsf{STorig})$. These optimisations do not eliminate atomic accesses, but can eliminate and move non atomic accesses in a wide variety of cases.

In a second time, we will look at the impact of the changes to the model, and try to determine when it is sound to eliminate atomic accesses.

Optimisations over opsemsets Because compilers like GCC or Clang have many distinct optimisation passes, we do not attempt to prove the correctness of specific source-to-source transformations. Instead, following Ševčík, we classify program transformations

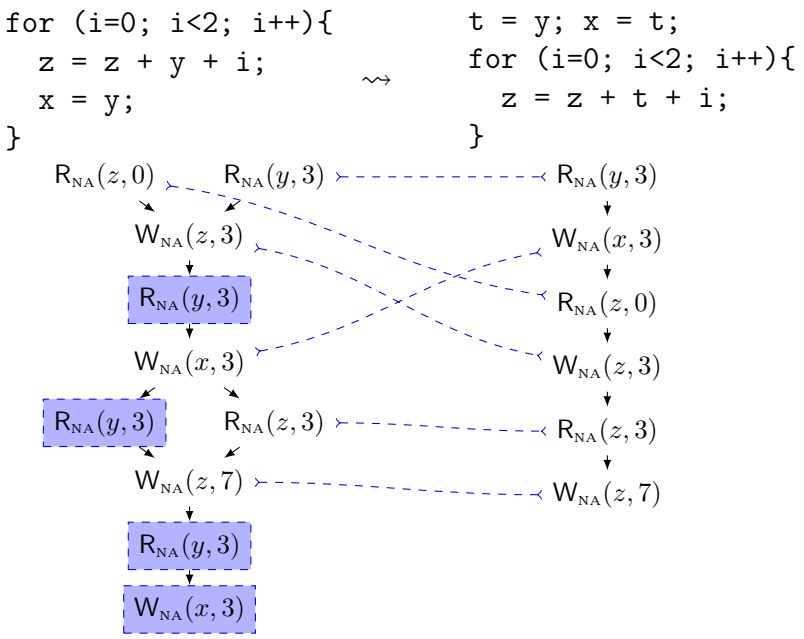


Figure 4.1: Example of the effect of a code transformation (Loop Invariant Code Motion) on an opsem (in which initially $y = 3$ and $z = 0$)

as *eliminations*, *reorderings*, and *introductions* over opsemsets (see Section 2.3.3 for the formal definition of opsemsets), and give a criterion under which they are correct in C11.

This allows us to abstract from all of the details of individual optimisations. Consider for example loop invariant code motion on the example of Figure 4.1. While the specific analysis to determine that code is a loop invariant can be very complex, we only care about the effect on opsems. In this specific example, some reads are eliminated because the value was already loaded (Read-after-Read elimination), and some non atomic accesses were reordered.

4.1 Optimising accesses around atomics

4.1.1 Eliminations of actions

We define semantic elimination and discuss informally its soundness criterion; we then state the soundness theorems and briefly describe the proof structure.

Definition 1. *An action is a* release *if it is an unlock action, an atomic write with memory-order* REL *or* SC*, a fence or read-modify-write with memory-order* REL*,* R/A *or* SC*.*

Semantically, release actions can be seen as potential sources of sw edges. The intuition is that they "release" permissions to access shared memory to other threads.

Definition 2. *An action is an* acquire *if it is a lock action, or an atomic read with memory-order* ACQ *or* SC*, or a fence or read-modify-write with memory order* ACQ*,* R/A *or* SC*.*

Acquire actions can be seen as potential targets of sw edges. The intuition is that they "acquire" permissions to access shared memory from other threads.

To simplify the presentation we omit dynamic thread creation. This is easily taken into account by stating that spawning a thread has release semantics, while the first accesses in SB-order of the spawned function have acquire semantics. Reciprocally, the last actions of a thread have release semantics and a thread-join has acquire semantics.

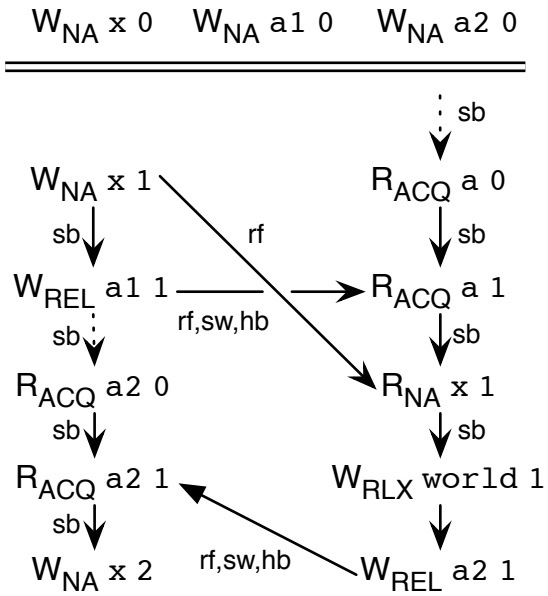
A key concept is that of a *same-thread release-acquire pair*:

Definition 3. *A* same-thread release-acquire pair *(shortened* st-release-acquire pair*) is a pair of actions (r, a) such that r is a release, a is an acquire, and $r <_{sb} a$.*

Note that these may be to different locations and never synchronise together. To understand the role they play in optimization soundness, consider the code on the left, running in the concurrent context on the right:

```
                                x=0; atomic a1,a2=0
x = 1;                      ||  while(0==a1.load(acq)) {};
a1.store(1,rel);            ||  printf("%i",x);
while(0==a2.load(acq)) {};  ||  a2.store(1,rel);
x = 2;                      ||
```

All executions have similar opsems and witnesses, depicted below (we omitted *rf* arrows from initialisation writes). No consistent execution has a race and the only observable behaviour is printing `1`.



Eliminating the first store to `x` (which might appear redundant as `x` is later overwritten by the same thread) would preserve DRF but would introduce a new behaviour where `0` is printed. However, if either the release or the acquire were not in between the two stores, then this context would be racy (respectively between the load performed by the print and the first store, or between the load and the second store) and it would be correct to optimize away the first write. More generally, the proof of the Theorem 4 below clearly shows that the presence of an intervening same-thread release-acquire is a necessary condition to allow a discriminating context to interact with a thread without introducing data races.

Definition 4. *A read action $a, t{:}R_{NA}\, l\, v$ is* eliminable *in an opsem O of the opsemset P if one of the following applies:*

Read after Read (RaR): there exists another action $r, t{:}\mathsf{R_{NA}}\, l\, v$ such that $r <_{sb} a$, and there does not exist a memory access to location l or a st-release-acquire pair SB-between r and a;

Read after Write (RaW): there exists an action $w, t{:}\mathsf{W_{NA}}\, l\, v$ such that $w <_{sb} a$, and there does not exist a memory access to location l or a st-release-acquire pair SB-between w and a;

Irrelevant Read (IR): for all values v' there exists an opsem $O' \in P$ and a bijection f between actions in O and actions in O', such that $f(a) = a', t{:}\mathsf{R_{NA}}\, l\, v'$, for all actions $u \in O$ different from a, $f(u) = u$, and f preserves SB and asw.

A write action $a, t{:}\mathsf{W_{NA}}\, l\, v$ is eliminable *in an opsem O of the opsemset P if one of the following applies:*

Write after Read (WaR): there exists an action $r, t{:}\mathsf{R}_l\, v$ such that $r <_{sb} a$, and there does not exist a memory access to location l or a st-release-acquire pair SB-between r and a;

Overwritten Write (OW): there exists another action $w, t{:}\mathsf{W_{NA}}\, l\, v'$ such that $a <_{sb} w$, and there does not exist a memory access to location l or a st-release-acquire pair SB-between a and w;

Write after Write (WaW): there exists another action $w, t{:}\mathsf{W_{NA}}\, l\, v$ such that $w <_{sb} a$, and there does not exist a memory access to location l or a st-release-acquire pair SB-between w and a.

Note that the OW rule is the only rule where the value can differ between the action eliminated and the action that justifies the elimination. The IR rule can be rephrased as "a read in an execution is irrelevant if the program admits other executions (one for each value) that only differ for the value returned by the irrelevant read".

Definition 5. *An opsem O' is an* elimination *of an opsem O if there exists a injection $f : O' \to O$ that preserves actions, sb, and asw, and such that the set $O \backslash f(O')$ contains exactly one eliminable action. The function f is called an* unelimination.

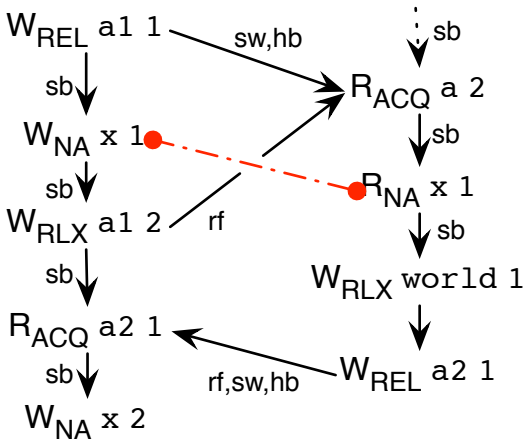
To simplify the proof of Theorem 4 the definition above allows only one elimination at a time (this avoids a critical pair between the rules OW and WaW whenever we have two writes of the same value to the same location), but, as the theorem shows, this definition can be iterated to eliminate several actions from one opsem while retaining soundness. The definition of eliminations lifts pointwise to opsemsets:

Definition 6. *An opsemset P' is an* elimination *of an opsemset P if for all opsem $O' \in P'$ there exists an opsem $O \in P$ such that O' is an elimination of O.*

You may remember from Section 2.3.3 that relaxed writes can create synchronisation when they follow a release fence or are part of a release sequence (see Figure 4.2 for an example, and Section 3.2.2 for a detailed discussion of this feature of the model).

These subtleties must be taken into account in the elimination correctness proof but do not invalidate the intervening same-thread release-acquire pair criterion. This follows from a property of the C11/C++11 design that makes every sw edge relate a release action to an acquire action. For instance, in the program below it is safe to remove the first write to x as OW, because all discriminating contexts will be necessarily racy:

```
a1.store(1,rel);
x = 1;
a1.store(2,rlx);
while(0==a2.load(acq)) {};
x = 2;
```



We establish that our semantic transformations have the following properties: any execution of the transformed opsemset has the same observable behaviour as some execution of the original opsemset, and the transformation preserves data-race freedom. As C11 and C++11 do not provide any guarantee for racy programs we cannot prove any result about out-of-thin-air value introduction.

Theorem 4. *Let the opsemset P' be an elimination of the opsemset P. If P is well defined, then so is P', and any execution of P' has the same observable behaviour as some execution of P.*

We sketch the structure of the proof; details can be found in Appendix A. Let the opsemset P' be an elimination of the opsemset P, and let (O', W') be an execution of P' (that is, a pair of an opsem and a witness). Since P' is an elimination of P, there is at least one opsem $O \in P$ such that O' is an elimination of O, and an unelimination function f that injects the events of the optimized opsem into the events of the unoptimized opsem. We build the mo and sc relations of the witness of O by lifting the mo' and sc' relations of the witness W': this is always possible because our program transformations do not alter any atomic access. Analogously it is possible to build the rf relation on atomic accesses by lifting the rf' one. To complete the construction of the witness we lift the rf' relation on non atomic events as well, and complete the rf relation following the case analysis below on the eliminated events in O:

- RaR: if i is a read action eliminated with the RaR rule because of a preceding read action r, and $w <_{rf} r$, then add $w <_{rf} i$;
- RaW: if i is a read action eliminated with the RaW rule because of a preceding write action w, then add $w <_{rf} i$;

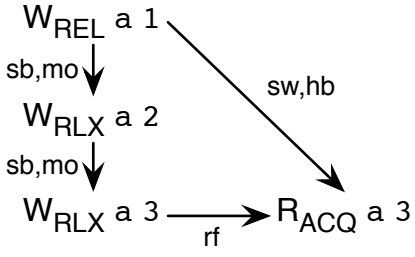


Figure 4.2: Example of a release sequence

- IR: if i is an irrelevant read, and there is a write event to the same location that happens before it, then let w be a write event to the same location maximal with respect to hb and add $w <_{rf} i$;
- OW: rf is unchanged;
- WaW: if i is a write event eliminated by the WaW rule because of the preceding write event w, then for all actions r such that $w <_{rf'} r$ and $i <_{\mathsf{hb}} r$, replace $w <_{rf} r$ by $i <_{rf} r$;
- WaR: if i is a read event eliminated by the WaR rule, then every read of the same value at the same location, that happens-after i and that either read from a write $w <_{\mathsf{hb}} i$ or does not read from any write, now reads from i.

This completes the construction of the witness W and in turn of the candidate execution (O, W) of P. We must now prove that (O, W) is consistent, in particular that it satisfies *consistent non atomic read values*, for which the construction has been tailored. This proceeds by a long case disjunction that relies on the following constructions:

- the absence of a release-acquire pair between two accesses a and b in the same thread guarantees the absence of an access c in another thread with $a <_{\mathsf{hb}} c <_{\mathsf{hb}} b$.
- in some cases the candidate execution (O, W) turns out to have conflicting accesses a and b that are not ordered by hb. We use the fact that opsemsets are receptive and closed under SB-prefix to build another candidate execution of P where a and b are still hb-unordered, but for which we can prove it is a consistent execution (not necessarily with the same observable behaviour). From this we deduce that P is not data-race free and ignore these cases.

By construction the consistent execution (O, W) has the same observable behaviour as (O', W'); we conclude by showing that (O', W') can not have undefined behaviours that (O, W) does not have.

4.1.2 Reorderings of actions

Most compiler optimizations do not limit themselves to eliminating memory accesses, but also reorder some of them. In this category fall all the code motion optimizations. Of course not all accesses are reorderable without introducing new behaviours. In this section we state and prove the correctness of the class of reorderings we have observed being performed by `gcc` and `clang`, ignoring more complex reordering schemes.

Definition 7. *Two actions a and b are* reorderable *if they access different memory locations and neither is a synchronisation action (that is a lock, unlock, atomic access, rmw, or fence action).*

Definition 8. *An opsem O' is a* reordering *of an opsem O if: (i) the set of actions in O and O' are equal; (ii) O and O' define the same asw; (iii) for all actions $a, b \in O$, if $a <_{sb} b$ then either $a <_{sb'} b$ or $b <_{sb'} a$ and a is reorderable with b.*

Like for eliminations, the definition of reorderings lifts pointwise to opsemsets:

Definition 9. *An opsemset P' is a* reordering *of an opsemset P if for all opsems $O' \in P'$ there exists an opsem $O \in P$ such that O' is a reordering of O.*

The soundness theorem for reorderings can be stated analogously to the one for eliminations and, although the details are different, the proofs follow the same structure. In particular the proof shows that, given a witness for an execution of a reordered opsem, it is possible to build the happens-before relation for the witness for the corresponding source opsem by lifting the synchronise-with relation, which is unchanged by the reorderings and relates the same release/acquire events in the two witnesses.

Theorem 5. *Let the opsemset P' be a reordering of the opsemset P. If P is well defined, then so is P', and any execution of P' has the same observable behaviour as some execution of P.*

As the SB relation is partial, reordering instructions in the source code can occasionally introduce SB arrows between reordered actions. For instance, the optimized trace of Figure 2.11 not only reorders the $\mathsf{W_{NA}\,x\,3}$ with the $\mathsf{R_{NA}\,z\,0}$ and $\mathsf{W_{NA}\,z\,3}$ actions but also introduces an SB arrow between the first two events $\mathsf{R_{NA}\,y\,3}$ and $\mathsf{R_{NA}\,z\,0}$. Unfortunately, as we saw in Section 3.1.2, adding such SB arrows is unsound in general. So we cannot directly prove the soundness of the optimisation in Figure 2.11

4.1.3 Introductions of actions

Even if it seems counterintuitive, compilers tend to introduce loads when optimizing code (introducing writes is incorrect in DRF models most of the time [BA08], and always dubious — see the final example in Section 5.3). Usually the introduced loads are irrelevant, that is their value is never used. This program transformation sounds harmless but it can introduce data races and does not lend itself to a theorem analogous to those for eliminations and reorderings. Worse than that, if combined with DRF-friendly optimizations it can introduce unexpected behaviours [Sev11]. We conjecture that a soundness property can be stated relating the source semantics to the actual hardware behaviour, along the lines of: if an opsem O' is obtained from an opsem O by introducing irrelevant reads, and it is then compiled naively following the standard compilation schemes for a given architecture [C11, Ter08, MS11], then all the behaviours observable on the architecture are allowed by the original opsem. In some cases the introduced loads are not irrelevant, but are RaR-eliminable or RaW-eliminable. We proved that RaR-eliminable and RaW-eliminable introductions preserve DRF and do not introduce new behaviours under the hypothesis that there is no *release* action in SB order between the introduced read and the action that justifies it.

4.2 Optimizing atomic accesses

In the previous section we focused on the optimisations that we observed being done by Clang and GCC; and on the C11 model as it is currently defined. We will now look in more details at the soundness of more optimisations, including eliminations of atomic accesses; and at how they interact with the fixes we proposed in Chapter 3.

4.2.1 Basic Metatheory of the Corrected C11 Models

In this section, we develop basic metatheory of the various corrections to the C11 model, which will assist us in verifying the program transformations in the next sections.

4.2.1.1 Semiconsistent Executions

We observe that in the monotone models (see Definition 12) the happens-before relation appears negatively in all axioms except for the $\Longrightarrow$ direction of the $\mathsf{ConsRFdom}$ axiom. It turns out, however, that this apparent lack of monotonicity with respect to happens-before does not cause problems as it can be circumvented by the following lemma.

Definition 10 (Semiconsistent Executions)**.** *An execution is* semiconsistent *with respect to a model M iff it satisfies all the axioms of the model except for the $\Longrightarrow$ direction of the* $\mathsf{ConsRFdom}$ *axiom.*

Lemma 1 (Semiconsistency)**.** *Given a semiconsistent execution (O, rf, mo, sc) with respect to M for $M \neq (\mathsf{ConsRFna}, \_, \_, \_)$, there exists $rf' \subseteq rf$ such that (O, rf', mo, sc) is consistent with respect to M.*

Proof. We pick rf' as the greatest fixed point of the functional:

$$F(rf)(x) := \begin{cases} rf(x) & \text{if } \exists y.\ \mathsf{hb}(y, x) \wedge \mathsf{iswrite}_{\mathsf{loc}(x)}(y) \\ \text{undefined} & \text{otherwise} \end{cases}$$

that is smaller than rf (with respect to $\subseteq$). Such a fixed point exists by Tarski's theorem as the function is monotone. By construction, it satisfies the $\mathsf{ConsRFdom}$ axiom, while all the other axioms follow easily because they are antimonotone in rf. □

4.2.1.2 Monotonicity

We move on to proving the most fundamental property of the corrected models: *monotonicity*, saying that if we weaken the access modes of some of the actions of a consistent execution and/or remove some sb edges, the execution remains consistent.

Definition 11 (Access type ordering)**.** *Let $\sqsubseteq : \mathcal{P}(MO \times MO)$ be the least reflexive and transitive relation containing* RLX $\sqsubseteq$ REL $\sqsubseteq$ REL-ACQ $\sqsubseteq$ SC*, and* RLX $\sqsubseteq$ ACQ $\sqsubseteq$ REL-ACQ*.*

We lift the access order to memory actions, $\sqsubseteq : \mathcal{P}(EV \times EV)$, by letting $act \sqsubseteq act$, $\mathsf{R}_X(\ell, v) \sqsubseteq \mathsf{R}_{X'}(\ell, v)$, $\mathsf{W}_X(\ell, v) \sqsubseteq \mathsf{W}_{X'}(\ell, v)$, $\mathsf{C}_X(\ell, v, v') \sqsubseteq \mathsf{C}_{X'}(\ell, v, v')$, $\mathsf{F}_X \sqsubseteq \mathsf{F}_{X'}$, and $\mathsf{skip} \sqsubseteq \mathsf{F}_{X'}$, whenever $X \sqsubseteq X'$. We also lift this order to functions pointwise: $lab \sqsubseteq lab'$ iff $\forall a.\ lab(a) \sqsubseteq lab'(a)$.

Monotonicity does not hold for all the models we consider, but only after some necessary fixes have been applied. We call those corrected models monotone.

Definition 12. *We call a memory model, M,* monotone*, if and only if $M \neq (\mathsf{ConsRFna}, \_, \_, \_)$ and $M \neq (\_, \mathsf{SCorig}, \_, \_)$.*

Theorem 6 (Monotonicity)**.** *For a monotone memory model M, if $\mathsf{Consistent}_M(lab, sb, asw, rf, mo, sc)$ and $lab' \sqsubseteq lab$ and $sb' \subseteq sb$, then there exist $rf' \subseteq rf$ and $sc' \subseteq sc$ such that $\mathsf{Consistent}_M(lab', sb', asw, rf', mo, sc')$.*

This theorem approximately says that (for monotone models) for every execution obtained by adding edges to sb' (creating sb) and by adding elements to lab' (creating lab), we can find an execution of the original program with the same behavior.

Proof sketch. From Lemma 1, it suffices to prove that the execution $(lab', sb', asw, rf, mo, sc')$ is semiconsistent. We can show this by picking:

$$sc'(x, y) \stackrel{\mathsf{def}}{=} sc(x, y) \wedge \mathsf{isSC}(lab'(x)) \wedge \mathsf{isSC}(lab'(y))$$

We can show that $\mathsf{hb}' \subseteq \mathsf{hb}$, and then all the axioms of the model follow straightforwardly. □

From Theorem 6, we can immediately show the soundness of three simple kinds of program transformations:

- *Expression evaluation order linearisation* and *sequentialisation*, because in effect they just add *sb* edges to the program;
- *Strengthening of the memory access orders*, such as replacing a relaxed load by an acquire load; and
- *Fence insertion*, because this can be seen as replacing a skip node (an empty fence) by a stronger fence.

4.2.1.3 Prefixes of Consistent Executions

Another basic property we would like to hold for a memory model is for any prefix of a consistent execution to also form a consistent execution. Such a property would allow, for instance, to execute programs in a stepwise operational fashion generating the set of consistent executions along the way. It is also very useful in proving the DRF theorem and the validity of certain optimizations by demonstrating an alternative execution prefix of the program that contradicts the assumptions of the statement to be proved (e.g., by containing a race).

One question remains: Under which relation should we be considering execution prefixes? To make the result most widely applicable, we want to make the relation as small as possible, but at the very least we must include (the dependent part of) the program order, *sb* and *asw*, in order to preserve the program semantics, as well as the reads from relation, *rf*, in order to preserve the memory semantics. Moreover, in the case of RSorig models, as shown in the example from Section 3.2.2, we must also include *mo*-prefixes.

Definition 13 (Prefix closure)**.** *We say that a relation, R, is* prefix closed *on a set, S, iff* $\forall a, b.\ R(a, b) \wedge b \in S \implies a \in S$.

Definition 14 (Prefix opsem)**.** *An opsem (lab', sb', asw') is a* prefix *of another opsem (lab, sb, asw) iff $lab' \subseteq lab$, $sb' = sb \cap (\mathsf{dom}(lab') \times \mathsf{dom}(lab'))$, $asw' = asw \cap (\mathsf{dom}(lab') \times \mathsf{dom}(lab'))$, and sb and asw are prefix closed on $\mathsf{dom}(lab')$.*

Theorem 7. *Given a model M, opsems $O = (lab, \_, \_)$ and $O' = (lab', \_, \_)$ and a witness $W = (rf, mo, sc)$, if $\mathsf{Consistent}_M(O, W)$ and O' is a prefix of O and $\{(a, b) \mid rf(b) = a\}$ is prefix-closed on $\mathsf{dom}(lab')$ and either $M = (\_, \_, \mathsf{RSnew}, \_)$ or mo is prefix-closed on $\mathsf{dom}(lab')$, then there exists W' such that $\mathsf{Consistent}_M(O', W')$.*

Proof (sketch). We pick W' to be W restricted to the actions in $\mathsf{dom}(lab')$. Then, we show $\mathsf{hb}' = \mathsf{hb} \cap (\mathsf{dom}(lab') \times \mathsf{dom}(lab'))$ and that each consistency axiom is preserved. □

To be able to use such a theorem in proofs, the relation defining prefixes should be acyclic. This is because we would like there to exist a maximal element in the relation, which we can remove from the execution and have the resulting execution remain consistent. This means that, for example, in the Arf model, we may want to choose $\mathsf{hb} \cup rf$ as our relation. Unfortunately, however, this does not quite work in the RSorig model and requires switching to the RSnew model.

4.2.2 Verifying Instruction Reorderings

$\downarrow a \setminus b \rightarrow$	$\mathsf{R}_{\mathrm{NA}\mid\mathrm{RLX}\mid\mathrm{ACQ}}(\ell')$	$\mathsf{R}_{\mathrm{SC}}(\ell')$	$\mathsf{W}_{\mathrm{NA}}(\ell')$	$\mathsf{W}_{\mathrm{RLX}}(\ell')$	$\mathsf{W}_{\mathrm{REL}\mid\mathrm{SC}}(\ell')$	$\mathsf{C}_{\mathrm{RLX}\mid\mathrm{ACQ}}(\ell')$	$\mathsf{C}_{\sqsupseteq\mathrm{REL}}(\ell')$	$\mathsf{F}_{\mathrm{ACQ}}$	$\mathsf{F}_{\mathrm{REL}}$
$\mathsf{R}_{\mathrm{NA}}(\ell)$	✓Thm.8	✓Thm.8	✓Thm.8/?/✗4.2.3.3	✓Thm.8/?/✗4.2.3.3	✗4.2.3.1	✓Thm.8/?/✗4.2.3.3	✗4.2.3.1	✓Thm.8	✗4.2.3.1
$\mathsf{R}_{\mathrm{RLX}}(\ell)$	✓Thm.8	✓Thm.8	✓Thm.8/?/✗4.2.3.3	✓Thm.8/✗4.2.3.2/✗4.2.3.3	✗4.2.3.1	✓Thm.8/✗4.2.3.2/✗4.2.3.3	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1
$\mathsf{R}_{\mathrm{ACQ}\mid\mathrm{SC}}(\ell)$	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✓Thm.9	✗4.2.3.1
$\mathsf{W}_{\mathrm{NA}\mid\mathrm{RLX}\mid\mathrm{REL}}(\ell)$	✓Thm.8	✓Thm.8	✓Thm.8	✓Thm.8	✗4.2.3.1	✓Thm.8	✗4.2.3.1	✓Thm.8	✗4.2.3.1
$\mathsf{W}_{\mathrm{SC}}(\ell)$	✓Thm.8	✗4.2.3.1	✓Thm.8	✓Thm.8	✗4.2.3.1	✓Thm.8	✗4.2.3.1	✓Thm.8	✗4.2.3.1
$\mathsf{C}_{\mathrm{RLX}\mid\mathrm{REL}}(\ell)$	✓Thm.8	✓Thm.8	✓Thm.8/?/✗4.2.3.3	✓Thm.8/✗4.2.3.2/✗4.2.3.3	✗4.2.3.1	✓Thm.8/✗4.2.3.2/✗4.2.3.3	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1
$\mathsf{C}_{\sqsupseteq\mathrm{ACQ}}(\ell)$	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✓Thm.9	✗4.2.3.1
$\mathsf{F}_{\mathrm{ACQ}}$	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	✗4.2.3.1	=	✗4.2.3.1
$\mathsf{F}_{\mathrm{REL}}$	✓Thm.8	✓Thm.8	✓Thm.8	✗4.2.3.1	✓Thm.10	✗4.2.3.1	✓Thm.10	✓Thm.8	=

Table 4.1: Allowed parallelisations $a\,;b \rightsquigarrow a \parallel b$ in monotone models, and therefore reorderings $a\,;b \rightsquigarrow b\,;a$. We assume $\ell \neq \ell'$. Where multiple entries are given, these correspond to Naive/Arf/Arfna. Ticks cite the appropriate theorem, crosses the counterexample (in Section 4.2.3). Question marks correspond to unknown cases. (We conjecture these are valid, but need a more elaborate definition of opsem prefixes to prove.)

We proceed to the main technical results of this chapter, namely the proofs of validity for the various program transformations. Having already discussed the simple monotonicity-based ones, we now focus on transformations that reorder adjacent instructions that do not access the same location.

We observe that for monotone models, a reordering can be decomposed into a parallelisation followed by a linearisation:

$$a\,;b \xrightarrow{\text{under some conditions}} a \parallel b \xrightarrow{\text{By Theorem 6}} b\,;a$$

We summarise the allowed reorderings/parallelisations in Table 4.1. There are two types of allowed updates:

(§4.2.2.1) "Roach motel" instruction reorderings, and

(§4.2.2.2) Fence reorderings against the roach motel semantics.

For the negative cases, we provide counterexamples in Section 4.2.3.

4.2.2.1 Roach Motel Instruction Reorderings

The "roach motel" reorderings are the majority among those in Table 4.1 and are annotated by '✓Thm.8.' They correspond to moving accesses into critical sections, i.e. sinking them past acquire accesses, or hoisting them above release accesses. The name comes from the saying "roaches check in but do not check out".

This category contains all reorderable pairs of actions, a and b, that are adjacent according to *sb* and *asw*. We say that two actions a and b are *adjacent* according to a relation R if (1) every action directly reachable from b is directly reachable from a; (2) every action directly reachable from a, except for b, is also directly reachable by b; (3) every action that reaches a directly can also reach b directly; and (4) every action that reaches b directly, except for a, can also reach a directly. Note that adjacent actions are not necessarily related by R.

Definition 15 (Adjacent actions)**.** *Two actions a and b are* adjacent *in a relation R, written $\mathsf{Adj}(R, a, b)$, if for all c, we have:*
(1) $R(b,c) \Rightarrow R(a,c)$*, and (2)* $R(a,c) \wedge c \neq b \Rightarrow R(b,c)$*, and*
(3) $R(c,a) \Rightarrow R(c,b)$*, and (4)* $R(c,b) \wedge c \neq a \Rightarrow R(c,a)$*.*

Two actions a and b are *reorderable* if (1) they belong to the same thread; (2) they do not access the same location, (3) a is not an acquire access or fence, (4) b is not a release access or fence, (5) if the model is based on Arfna or Arf and a is a read, then b is not a write, (6) if a is a release fence, then b is not an atomic write, (7) if b is an acquire fence, then a is not an atomic read, and (8) a and b are not both SC actions.

Definition 16 (Reorderable pair)**.** *Two distinct actions a and b are* reorderable *in a memory model M, written $\mathsf{Reord}_M(a,b)$, if*
(1) $\mathsf{tid}(a) = \mathsf{tid}(b)$
and (2) $\mathsf{loc}(a) \neq \mathsf{loc}(b)$
and (3) $\neg\mathsf{isAcq}(a)$
and (4) $\neg\mathsf{isRel}(b)$
and (5i) $\neg(M = (\mathsf{Arfna}, \_, \_, \_) \wedge \mathsf{isread}(a) \wedge \mathsf{iswrite}(b))$
and (5ii) $\neg(M = (\mathsf{Arf}, \_, \_, \_) \wedge \mathsf{isread}(a) \wedge \mathsf{iswrite}(b))$
and (6) $\neg(\mathsf{isFenceRel}(a) \wedge \mathsf{isAtomicWrite}(b))$
and (7) $\neg(\mathsf{isAtomicRead}(a) \wedge \mathsf{isFenceAcq}(b))$
and (8) $\neg(\mathsf{isSC}(a) \wedge \mathsf{isSC}(b))$

Theorem 8. *For a monotone M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(sb, a, b)$, $\mathsf{Adj}(asw, a, b)$, *and* $\mathsf{Reord}_M(a, b)$, *there exists* W',
(i) $\mathsf{Consistent}_M(lab, sb \cup \{(a, b)\}, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, *and*
(iii) $\mathsf{Racy}_M(lab, sb, W) \Rightarrow \mathsf{Racy}_M(lab, sb \cup \{(a, b)\}, W')$.

Proof (sketch). By Lemma 1, it suffices to show semiconsistency. The main part is then proving that $\mathsf{hb} = \mathsf{hb}' \cup \{(a, b)\}$, where hb (resp. hb') denotes the happens-before relation in $(lab, sb \cup \{(a, b)\}, asw, W)$ (resp. (lab, sb, asw, W)). Hence these transformations do not really affect the behaviour of the program, and the preservation of each axiom is a simple corollary. □

The proof of Theorem 8 (and similarly those of Theorems 9 and 10 in Section 4.2.2.2), require only conditions (1) and (3) from the definition of adjacent actions; Conditions (2) and (4) are, however, important for the theorems of Section 4.2.4.1, and so, for simplicity, we presented a single definition of when two actions are adjacent.

4.2.2.2 Non-RM Reorderings with Fences

The second class is comprised of a few valid reorderings between a fence and a memory access of the same or stronger type. In contrast to the previous set of transformations, these new ones remove some synchronisation edges but only to fence instructions. As fences do not access any data, there are no axioms constraining these incoming and outgoing synchronisation edges to and from fences, and hence they can be safely removed.

Theorem 9. *For a monotone M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(sb, a, b)$, $\mathsf{Adj}(asw, a, b)$, $\mathsf{isAcq}(a)$, *and* $lab(b) = \mathsf{F}_{\text{ACQ}}$, *then*
(i) $\mathsf{Consistent}_M(lab, sb \cup \{(a, b)\}, asw, W)$ *and*
(ii) $\mathsf{Racy}_M(lab, sb, W) \Rightarrow \mathsf{Racy}_M(lab, sb \cup \{(a, b)\}, W)$.

Theorem 10. *For a monotone M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(sb, a, b)$, $\mathsf{Adj}(asw, a, b)$, $lab(a) = \mathsf{F}_{\text{REL}}$ *and* $\mathsf{isRel}(b)$, *then*
(i) $\mathsf{Consistent}_M(lab, sb \cup \{(a, b)\}, asw, W)$ *and*
(ii) $\mathsf{Racy}_M(lab, sb, W) \Rightarrow \mathsf{Racy}_M(lab, sb \cup \{(a, b)\}, W)$.

That is, we can reorder an acquire command over an acquire fence, and a release fence over a release command.

$$\mathsf{acq}\,;\mathsf{F}_{\text{ACQ}} \xrightarrow{\text{Thm. 9 \& 6}} \mathsf{F}_{\text{ACQ}}\,;\mathsf{acq} \qquad \mathsf{F}_{\text{REL}}\,;\mathsf{rel} \xrightarrow{\text{Thm. 10 \& 6}} \mathsf{rel}\,;\mathsf{F}_{\text{REL}}$$

4.2.3 Counter-examples for unsound reorderings

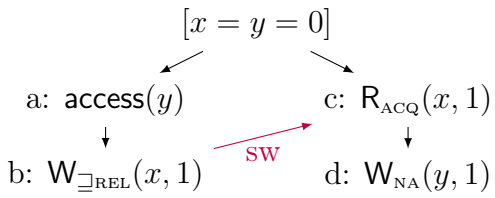
4.2.3.1 Unsafe Reordering Examples for All Models

We present a series of counterexamples showing that certain adjacent instruction permutations are invalid.

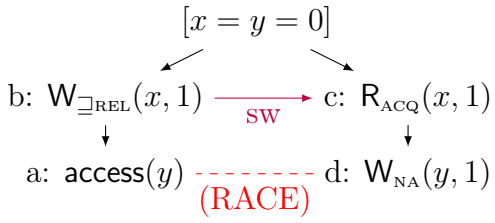
$\mathsf{access}; \mathsf{W}_{\mathbf{REL|SC}} \rightsquigarrow \mathsf{W}_{\mathbf{REL|SC}}; \mathsf{access}$ Our first counterexample shows that we cannot reorder a memory access past a release store. Consider the following program:

$$\begin{array}{l} \mathsf{access}(y); \\ x.\mathsf{store}(1, \sqsupseteq\text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ})) \\ \quad y = 1; \end{array}$$

This program is race-free as illustrated by the following execution:



If we reorder the instructions in the first thread, the following racy execution is possible.

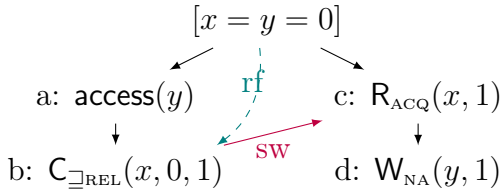


This means that the reordering introduced new behaviour and is therefore unsound.

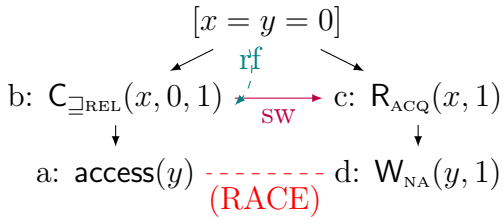
access; $\mathsf{C}_{\sqsupseteq\text{REL}} \rightsquigarrow \mathsf{C}_{\sqsupseteq\text{REL}}$; access We can construct a similar example to show that reordering an access past a release RMW is also unsound. Consider the program:

$$\begin{array}{l} \text{access}(y); \\ x.\mathsf{CAS}(0, 1, \sqsupseteq\text{REL}); \end{array} \Bigg\| \begin{array}{l} \text{if } (x.\text{load}(\text{ACQ})) \\ \quad y = 1; \end{array}$$

This program is race free because $y = 1$ can happen only after the load of x reads 1 and synchronises with the successful CAS.



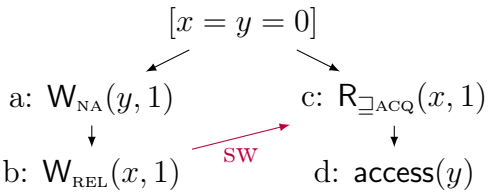
If we reorder the instructions in the first thread, the following racy execution is possible, and therefore the transformation is unsound.



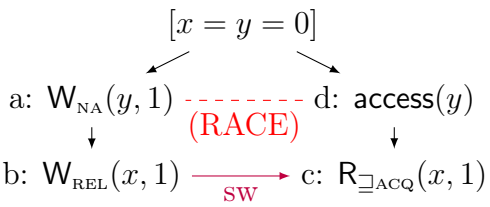
$\mathsf{R}_{\sqsupseteq\text{ACQ}}$; access $\rightsquigarrow$ access; $\mathsf{R}_{\sqsupseteq\text{ACQ}}$ Next, we construct dual examples showing that reordering an acquire read past a memory access is unsound. Consider the following example, where on the second thread, we use a busy loop to wait until the store of x has happened.

$$\begin{array}{l} y = 1; \\ x.\text{store}(1, \text{REL}); \end{array} \Bigg\| \begin{array}{l} \text{while } (1 \neq x.\text{load}(\sqsupseteq\text{ACQ})) \\ \text{access}(y); \end{array}$$

The program is race-free as demostrated by the following execution:



Reordering the $\mathsf{access}(y)$ above the load of x is unsound, because then the following execution would be possible:



This execution contains racy accesses to y, and therefore exhibits more behaviours than the original program does.

This shows that reordering an acquire load past a memory access is unsound. In a similar fashion, we can show that reordering an acquire RMW past a memory access is also unsound. All we have to do is consider the following program:

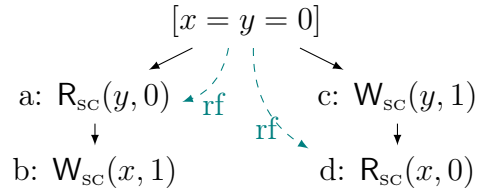
$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \;\Big\|\; \begin{array}{l} \mathsf{while}\ (\neg x.\mathsf{CAS}(1, 2, \sqsupseteq\text{ACQ})); \\ \mathsf{access}(y); \end{array}$$

where we have replaced the acquire load by an acquire CAS.

$\mathsf{W_{SC}}; \mathsf{R_{SC}} \rightsquigarrow \mathsf{R_{SC}}; \mathsf{W_{SC}}$ Consider the store-buffering program with SC accesses:

$$\begin{array}{l} x.\mathsf{store}(1, \text{SC}); \\ r_1 = y.\mathsf{load}(\text{SC}); \end{array} \;\Big\|\; \begin{array}{l} y.\mathsf{store}(1, \text{SC}); \\ r_2 = x.\mathsf{load}(\text{SC}); \end{array}$$

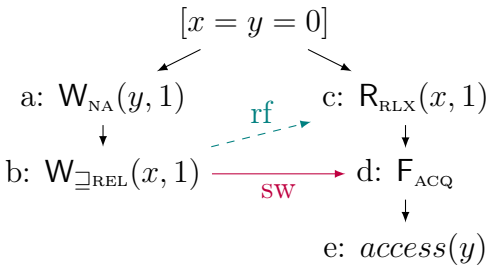
Since SC accesses are totally ordered, the outcome $r_1 = r_2 = 0$ is not possible. This outcome, however, can be obtained by reordering the actions of the left thread.



$\mathsf{F_{ACQ}}; \mathsf{access} \rightsquigarrow \mathsf{access}; \mathsf{F_{ACQ}}$ Consider the following program

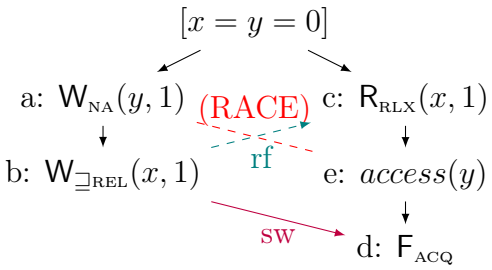
$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \;\Big\|\; \begin{array}{l} \mathsf{while}\ (1 \neq x.\mathsf{load}(\text{RLX})); \\ \mathsf{fence}(\text{ACQ}); \\ \mathsf{access}(y); \end{array}$$

The execution trace is as follows



In this execution $(a, e) \in hb$ and hence the execution is data race free.

The reordering $d; e \rightsquigarrow e; d$ would result in following execution



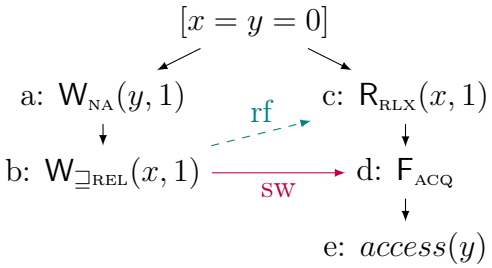
In this case the $(a, e) \notin hb$ and hence result in data race.

Hence the transformation is unsafe.

$\mathsf{R}_{\mathbf{RLX}}; \mathsf{F}_{\mathbf{ACQ}} \rightsquigarrow \mathsf{F}_{\mathbf{ACQ}}; \mathsf{R}_{\mathbf{RLX}}$ Consider the following program

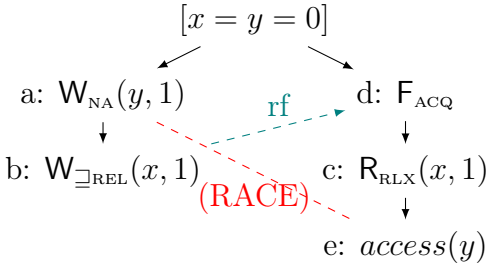
$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{while}\ (1 \neq x.\mathsf{load}(\text{RLX})); \\ \mathsf{fence}(\text{ACQ}); \\ \mathsf{access}(y); \end{array}$$

This program is well synchronised, because for the access to y to occur, the relaxed load must have read from the release store of x, and therefore the release store and the acquire fence synchronise.



In this execution $(a, e) \in \mathsf{hb}$ and hence the execution is data race free.

The reordering $c; d \rightsquigarrow d; c$ would result in following execution



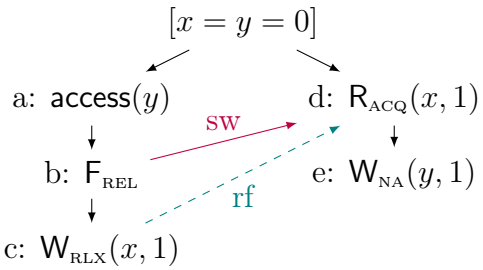
In this case $(a, e) \notin hb$ and hence there is a data race. Therefore, the transformation is unsafe.

With a similar program, we can show that moving a RLX or REL RMW past an acquire fence is also unsafe. (Just replace $1 \neq x.\mathsf{load}(\text{RLX})$ with $\neg x.\mathsf{CAS}(1, 2, \{\text{RLX}, \text{REL}\})$.)

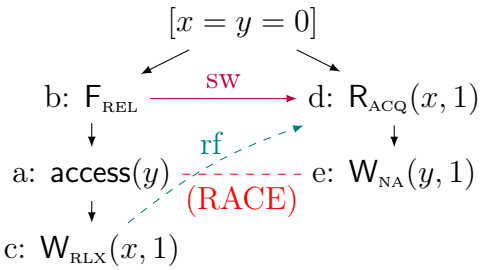
$\mathsf{access}; \mathsf{F}_{\mathbf{REL}} \rightsquigarrow \mathsf{F}_{\mathbf{REL}}; \mathsf{access}$ Similarly consider the program below

$$\begin{array}{l} \mathsf{access}(y); \\ \mathsf{fence}(\text{REL}); \\ x.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ})) \\ \quad y = 1; \end{array}$$

This program is data race free, because for a store of y to occur, the load of x must read 1 and therefore synchronize with the fence:



Reordering $a; b \rightsquigarrow b; a$, however, results in a racy execution:

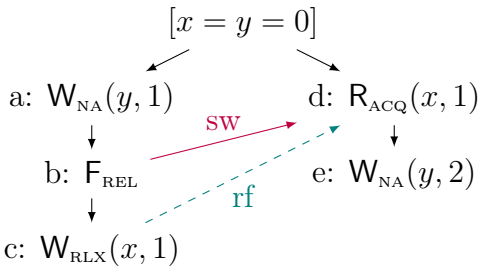


Therefore, the transformation is unsafe.

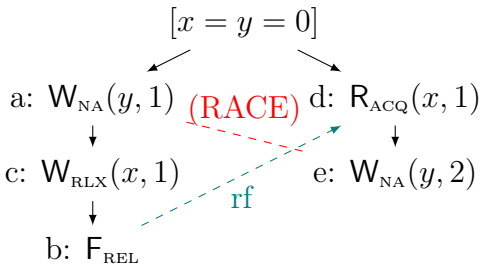
$\mathsf{F}_{\mathbf{REL}}; \mathsf{W}_{\mathbf{RLX}} \rightsquigarrow \mathsf{W}_{\mathbf{RLX}}; \mathsf{F}_{\mathbf{REL}}$ Consider the following program

$$\begin{array}{l} y = 1; \\ \mathsf{fence}(\text{REL}); \\ x.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ}))\ y = 2; \end{array}$$

The program is race-free because if $y = 2$ happens, then it happens after the $y = 1$ store because the load of x must synchronise with the fence.



Reordering $b; c \rightsquigarrow c; b$, however, can result in the following racy execution:



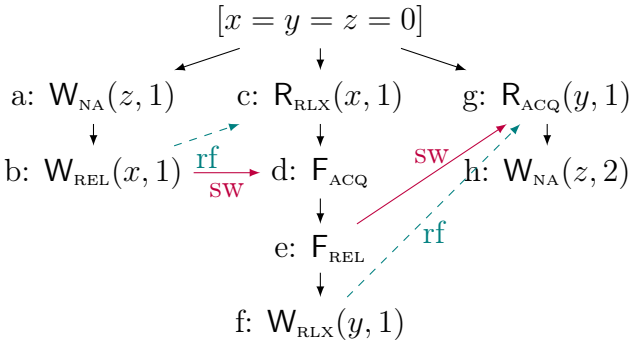
Hence, the transformation is unsafe.

With a similar program, we can show that moving a release fence past a RLX or ACQ RMW is also unsafe. (Just replace $x.\mathsf{store}(1, \text{RLX})$ with $x.\mathsf{CAS}(0, 1, \{\text{RLX}, \text{ACQ}\})$.)

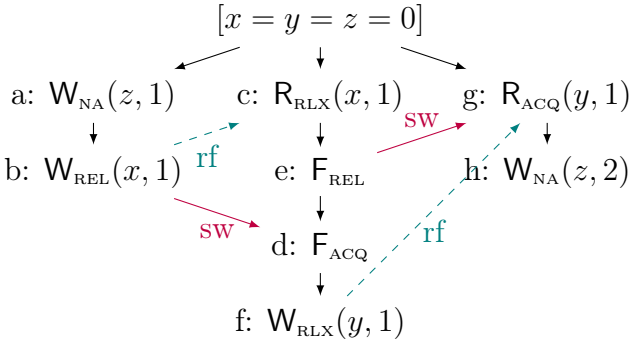
$\mathsf{F}_{\mathbf{ACQ}}; \mathsf{F}_{\mathbf{REL}} \rightsquigarrow \mathsf{F}_{\mathbf{REL}}; \mathsf{F}_{\mathbf{ACQ}}$ Consider the following program

$$
\begin{array}{l}
z = 1; \\
x.\mathsf{store}(1, \text{REL});
\end{array}
\;\Big\|\;
\begin{array}{l}
\mathsf{if}\ (x.\mathsf{load}(\text{RLX}))\ \{ \\
\quad \mathsf{fence}(\text{ACQ}); \\
\quad \mathsf{fence}(\text{REL}); \\
\quad y.\mathsf{store}(1, \text{RLX}); \\
\}
\end{array}
\;\Big\|\;
\begin{array}{l}
\mathsf{if}\ (y.\mathsf{load}(\text{ACQ})) \\
\quad z = 2;
\end{array}
$$

The program is race free because the only consistent execution containing conflicting accesses is the following:



which is, however, not racy because $\mathsf{hb}(a, h)$. Reordering d and e does, however, result in the following racy execution and is, therefore, unsound.

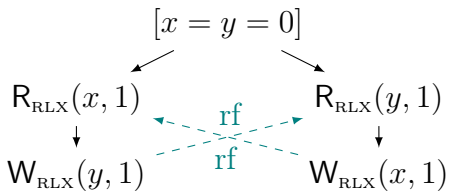


4.2.3.2 Counterexamples for the Arf model

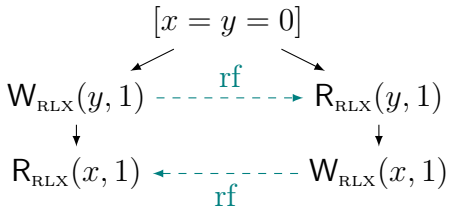
Reordering an atomic read past an adjacent atomic write is unsound in the Arf memory model. We show this first for an atomic load and an atomic store. Consider the following program, where implicitly all variables are initialised to 0.

$$\begin{array}{l} r = x.\mathsf{load}(\sqsupseteq \mathrm{RLX}); \\ y.\mathsf{store}(1, \sqsupseteq \mathrm{RLX}); \end{array} \;\Big\|\; \begin{array}{l} r' = y.\mathsf{load}(\mathrm{RLX}); \\ x.\mathsf{store}(1, \mathrm{RLX}); \end{array}$$

In this code the outcome $r = r' = 1$ is not possible. The only execution that could yield this result is the following,



which is inconsistent. If, however, we permute the instructions of the first thread, then this outcome is possible by the following execution:



Note that if we replace the load of x with a compare and swap, $x.\mathsf{CAS}(1, 2, \_)$, and/or the store of y with a compare and swap, $y.\mathsf{CAS}(0, 1, \_)$, the displayed source execution remains inconsistent, and while the target execution is valid. Hence reordering any kind of atomic read over any kind of atomic write is unsound in this model.

4.2.3.3 Counterexamples for the Arfna model

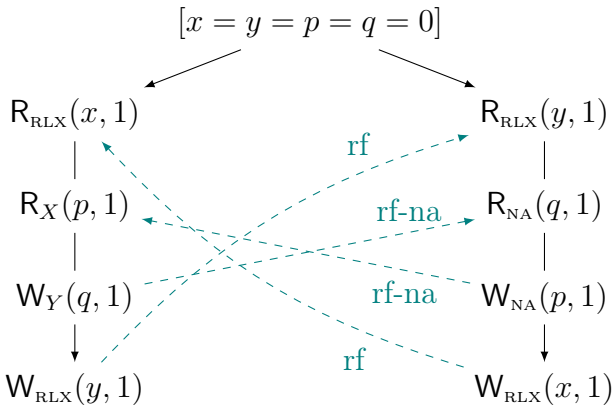
We show that reodering a non atomic or atomic read past an adjacent non atomic or atomic write is unsound. Consider the following program, where implicitly all variables are initialized to 0.

$$\begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\mathrm{RLX}))\ \{ \\ \quad t = p.\mathsf{load}(X); \\ \quad q.\mathsf{store}(1, Y); \\ \quad \mathsf{if}\ (t)\ y.\mathsf{store}(1, \mathrm{RLX}); \\ \} \end{array} \;\Big\|\; \begin{array}{l} \mathsf{if}\ (y.\mathsf{load}(\mathrm{RLX})) \\ \quad \mathsf{if}\ (q)\ \{ \\ \quad\quad p = 1; \\ \quad\quad x.\mathsf{store}(1, \mathrm{RLX}); \\ \quad \} \end{array}$$

(Where X and Y stand for any atomic or non atomic access modes.)

Note that this program is race-free and its only possible outcome is $p = q = 0$. (The racy execution yielding $p = q = 1$ is inconsistent because it contains a cycle forbidden by

the Arfna axiom.)

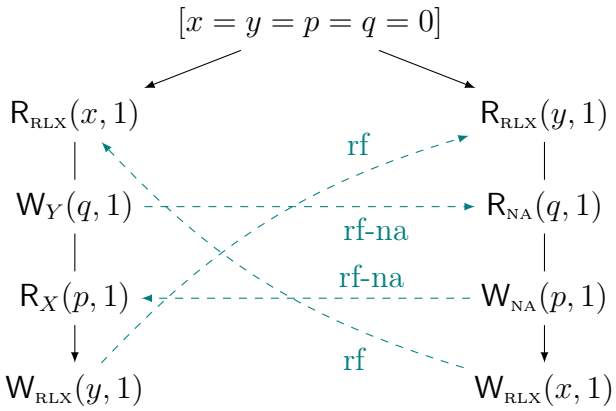


(The reads in this cycle are annotated by "rf-na.")

If, however, we permute the two adjacent non atomic access of the first thread as follows:

$$t = p; q = 1; \rightsquigarrow q = 1; t = p;$$

then the following racy execution:



is consistent, and therefore the target program has more behaviours than the source program.

Note that if we replace the load $p.\mathsf{load}(X)$ with a compare and swap, $p.\mathsf{CAS}(1, 2, X)$, and/or the $q.\mathsf{store}(1, Y)$ with a compare and swap, $q.\mathsf{CAS}(0, 1, Y)$, the displayed source execution remains inconsistent, and while the target execution is valid. Hence the transformation adds new behaviour and is unsound.

4.2.4 Verifying Instruction Eliminations

Next, we consider eliminating redundant memory accesses, as would be performed by standard optimizations such as common subexpression elimination or constant propagation. To simplify the presentation (and the proofs), in §4.2.4.1, we first focus on the cases where eliminating an instruction is justified by an adjacent instruction (e.g., a repeated read, or an immediately overwritten write). In §4.2.4.2, we will then tackle the general case.

4.2.4.1 Elimination of Redundant Adjacent Accesses

Repeated Read. The first transformation we consider is eliminating the second of two identical adjacent loads from the same location. Informally, if two loads from the same location are adjacent in program order, it is possible that both loads return the value written by the same store. Therefore, if the loads also have the same access type, the additional load will not introduce any new synchronisation, and hence we can always remove one of them, say the second.

$$\mathsf{R}_X(\ell, v)\,;\mathsf{R}_X(\ell, v) \rightsquigarrow \mathsf{R}_X(\ell, v)\,;\mathsf{skip} \tag{RAR-adj}$$

Formally, we say that a and b are adjacent if a is sequenced before b and they are adjacent according to sb and asw. That is:

$$\mathsf{Adj}(a, b) \stackrel{\mathsf{def}}{=} sb(a, b) \wedge \mathsf{Adj}(sb, a, b) \wedge \mathsf{Adj}(asw, a, b)$$

We can prove the following theorem:

Theorem 11 (RaR-Adjacent)**.** *For a monotone memory model M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(a, b)$, $lab(a) = \mathsf{R}_X(\ell, v)$, $lab(b) = \mathsf{skip}$, *and* $lab' = lab[b := \mathsf{R}_X(\ell, v)]$, *then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, *and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$.

This says that any consistent execution of the target of the transformation can be mapped to one of the program prior to the transformation. To prove this theorem, we pick $rf' := rf[b \mapsto rf(a)]$ and extend the sc order to include the (a, b) edge in case $X = \text{SC}$.

Read after Write. Similarly, if a load immediately follows a store to the same location, then it is always possible for the load to get the value from that store. Therefore, it is always possible to remove the load.

$$\mathsf{W}_X(\ell, v)\,;\mathsf{R}_Y(\ell, v) \rightsquigarrow \mathsf{W}_X(\ell, v)\,;\mathsf{skip} \tag{RAW-adj}$$

Formally, we prove the following theorem:

Theorem 12 (RaW-Adjacent)**.** *For a monotone memory model M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(a, b)$, $lab(a) = \mathsf{W}_X(\ell, v)$, $lab(b) = \mathsf{skip}$, $lab' = lab[b := \mathsf{R}_Y(\ell, v)]$ *and either* $Y \neq \text{SC}$ *or* $M \neq (\_, \_, \mathsf{STorig}, \_)$, *then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, *and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$.

Overwritten Write. If two stores to the same location are adjacent in program order, it is possible that the first store is never read by any thread. So, if the stores have the same access type we can always remove the first one. That is, we can do the transformation:

$$\mathsf{W}_X(\ell, v')\,;\mathsf{W}_X(\ell, v) \rightsquigarrow \mathsf{skip}\,;\mathsf{W}_X(\ell, v) \tag{OW-adj}$$

To prove the correctness of the transformation, we prove the following theorem saying that any consistent execution of the target program corresponds to a consistent execution of the source program.

Consider the following program:

$$x = y = 0;$$

$$\begin{array}{l|l} y.\mathsf{store}(1, \text{RLX}); & \\ \mathsf{fence}(\text{REL}); & t_2 = x.\mathsf{CAS}(0, 1, \text{ACQ}); \\ t_1 = x.\mathsf{load}(\text{RLX}); & t_3 = y.\mathsf{load}(\text{RLX}); \\ x.\mathsf{store}(t_1, \text{RLX}); & \end{array}$$

$$t_4 = x.\mathsf{load}(\text{RLX});$$

The outcome $t_1 = t_2 = t_3 = 0 \wedge t_4 = 1$ is not possible. If, however, we remove the $x.\mathsf{store}(t_1, \text{RLX})$ then this outcome becomes possible.

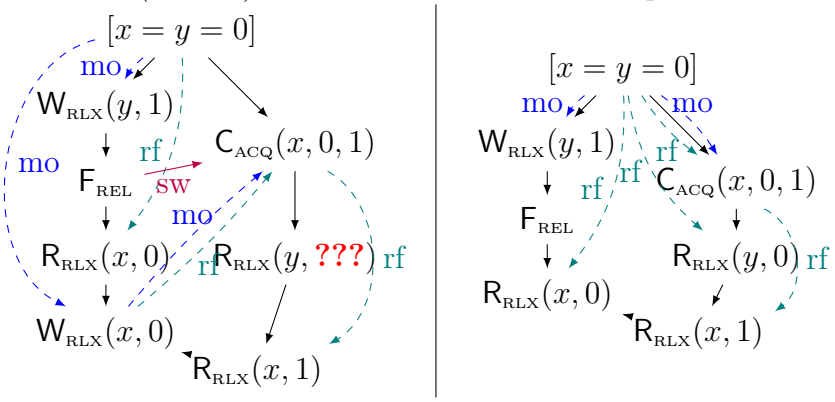


Figure 4.3: Counterexample for the 'write after read' elimination optimisation.

Theorem 13 (OW-Adjacent)**.** *For a monotone memory model M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$ *and* $\mathsf{Adj}(a, b)$ *and* $lab(a) = \mathsf{skip}$ *and* $lab(b) = \mathsf{W}_X(\ell, v)$ *and* $lab' = lab[a := \mathsf{W}_X(\ell, v')]$ *and* $\ell \neq \texttt{world}$*, then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$*,*
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$*, and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$*.*

Note that as a special case of this transformation, if the two stores are identical, we can alternatively remove the second one:

$$\mathsf{W}_X(\ell,v)\,;\mathsf{W}_X(\ell,v) \overset{\text{(OW-adj)}}{\rightsquigarrow} \mathsf{skip}\,;\mathsf{W}_X(\ell,v) \overset{\text{reorder}}{\rightsquigarrow} \mathsf{W}_X(\ell,v)\,;\mathsf{skip}$$

Write after Read. The next case to consider is what happens when a store immediately follows a load to the same location, and writes the same value as observed by the load.

$$\mathsf{R}_X(\ell, v)\,;\mathsf{W}_{\text{RLX}}(\ell, v) \rightsquigarrow \mathsf{R}_X(\ell, v)\,;\mathsf{skip}$$

In this case, can we eliminate the redundant store?

Well, actually, no, we cannot. Figure 4.3 shows a program demonstrating that the transformation is unsound. The program uses an atomic read-modify-write instruction, CAS, to update x, in parallel to the thread that reads x to be 0 and then writes back 0 to x.

Consider an execution in which the load of x reads 0 (enforced by $t_1 = 0$), the CAS succeeds (enforced by $t_2 = 0$) and is in modification order after the store to x (enforced by $t_4 = 1$ and the CohWR axiom). Then, because of the atomicity of CAS (axiom AtRMW), the CAS must read from the first thread's store to x, inducing a synchronisation edge

$$x = y = z = 0;$$

$y.\mathsf{store}(1, \text{RLX});$	$\Vert$	
$\mathsf{fence}(\text{REL});$	$\Vert$	$z.\mathsf{store}(1, \text{RLX});$
$t_1 = x.\mathsf{load}(\text{RLX});$	$\Vert$	$\mathsf{fence}(\text{SC});$
$x.\mathsf{store}(t_1, \text{RLX});$	$\Vert$	$t_3 = x.\mathsf{load}(\text{RLX});$
$\mathsf{fence}(\text{SC});$	$\Vert$	$t_4 = y.\mathsf{load}(\text{RLX});$
$t_2 = z.\mathsf{load}(\text{RLX});$	$\Vert$	

$$t_5 = x.\mathsf{load}(\text{RLX});$$

The outcome $t_1 = t_2 = t_3 = t_4 = 0 \wedge t_5 = 1$ is impossible. If, however, we remove the $x.\mathsf{store}(t1, \text{RLX})$ then this outcome becomes possible.

Figure 4.4: Counterexample for the 'write after read' elimination optimisation, with a sc fence instead of a CAS

between the two threads. As a result, by the CohWR axiom, the load of y cannot read the initial value (i.e., necessarily $t_3 \neq 0$).

If, however, we remove the store to x from the left thread, the outcome in question becomes possible as indicated by the second execution shown in Figure 4.3.

In essence, this transformation is unsound because we can force an operation to be ordered between the load and the store (according to the communication order). In the aforementioned counterexample, we achieved this by the atomicity of RMW instructions.

We can also construct a similar counterexample without RMW operations, by exploiting SC fences, as shown in Figure 4.4.

The above counterexamples rely on on the use of a release fence, and those only create SW edges in conjunction with atomic accesses. So they are not counterexamples to the Theorem 4.

4.2.4.2 Elimination of Redundant Non-Adjacent Operations

We proceed to the general case, where the removed redundant operation is in the same thread as the operation justifying its removal, but not necessarily adjacent to it.

In the appendix, we have proved three theorems generalising the theorems of Section 4.2.4.1. The general set up is that we consider two actions a and b in program order (i.e., $sb(a, b)$), accessing the same location ℓ (i.e., $\mathsf{loc}(a) = \mathsf{loc}(b) = \ell$), without any intermediate actions accessing the same location (i.e., $\nexists c.\ sb(a, c) \wedge sb(c, b) \wedge \mathsf{loc}(c) = \ell$). In addition, for the generalisations of Theorems 11 and 12 (respectively, of Theorem 13), we also require there to be no acquire (respectively, release) operation in between.

Under these conditions, we can reorder the action to be eliminated (using Theorem 8) past the intermediate actions to become adjacent to the justifying action, so that we can apply the adjacent elimination theorem. Then we can reorder the resulting "skip" node back to the place the eliminated operation was initially.

Chapter 5

Fuzzy-testing GCC and Clang for compiler concurrency bugs

5.1 Compiler concurrency bugs

We saw in the previous chapter that the criteria for correctness of compiler optimisations in C11 are subtle. Are they actually respected by compilers? In this chapter, we will first show an example of a GCC bug that appears only in a concurrent context, and then explain an approach for automatically detecting such bugs.

Consider the C program in Figure 5.1. Can we guess its output? This program spawns two threads executing the functions `th_1` and `th_2` and waits for them to terminate. The two threads share two global memory locations, `x` and `y`, but a careful reading of the code reveals that the inner loop of `th_1` is never executed and `y` is only accessed by the second thread, while `x` is only accessed by the first thread. As we saw in Chapter 2, such a program is called *data-race free*, and its semantics is defined as the interleavings of the actions of the two threads. So it should always print `42` on the standard output.

If we compile the above code with the version 4.7.0 of `gcc` on an x86_64 machine running Linux, and we enable some optimizations with the `-O2` flag (as in `g++ -std=c++11 -lpthread -O2 -S foo.c`) then, sometimes, the compiled code prints `0` to the standard output. This unexpected outcome is caused by a subtle bug in `gcc`'s implementation of the loop invariant code motion (LIM) optimization. If we inspect the generated assembly we discover that `gcc` saves and restores the content of `y`, causing `y` to be overwritten with `0`:

```
th_1:
    movl  x(%rip), %edx     # load x (1) into edx
    movl  y(%rip), %eax     # load y (0) into eax
    testl %edx, %edx        # if x != 0
    jne   .L2               #   jump to .L2
    movl  $0, y(%rip)
    ret
.L2:
    movl  %eax, y(%rip)    # store eax (0) into y
    xorl  %eax, %eax        # return 0 (NULL)
    ret
```

This optimization is sound in a sequential world because the extra store always rewrites

```
#include <stdio.h>
#include <pthread.h>

int x = 1; int y = 0;

void *th_1(void *p) {
  for (int l = 0; (l != 4); l++) {
    if (x) return NULL;
    for (y = 0; (y >= 26); ++y)
      ;
  }
}

void *th_2(void *p) {
  y = 42;
  printf("%d\n",y);
}

int main() {
  pthread_t th1, th2;
  pthread_create(&th1, NULL, th_1, NULL);
  pthread_create(&th2, NULL, th_2, NULL);
  (void)pthread_join (th1, NULL);
  (void)pthread_join (th2, NULL);
}
```

Figure 5.1: `foo.c`, a concurrent program miscompiled by `gcc` 4.7.0.

the initial value of `y` and the final state is unchanged. However, as we have seen, this optimization is unsound in the concurrent context provided by `th_2`, and the C11/C++11 standards forbid it. We have reported this bug, which has been fixed (see `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=52558`).

How to search for concurrency compiler bugs? We have a *compiler bug* whenever the code generated by a compiler exhibits a behaviour not allowed by the semantics of the source program. Differential random testing has proved successful at hunting compiler bugs. The idea is simple: a test harness generates random, *well defined*, source programs, compiles them using several compilers, runs the executables, and compares the outputs. The state of the art is represented by the Csmith tool by Yang, Chen, Eide and Regehr [YCER11], which over the last four years has discovered several hundred bugs in widely used compilers, including `gcc` and `clang`. However this work cannot find *concurrency compiler bugs* like the one we described above: despite being miscompiled, the code of `th_1` still has correct behaviour in a sequential setting.

A naive approach to extend differential random testing to concurrency bugs would be to generate *concurrent* random programs, compile them with different compilers, record *all* the possible outcomes of each program, and compare these sets. This works well in some settings, such as the generation and comparison of litmus tests to test hardware memory models; see for instance the work by Alglave et al. [AMSS11]. However this approach is unlikely to scale to the complexity of hunting C11/C++11 compiler bugs. Concurrent programs are inherently *non deterministic* and optimizers can compile away non determism. In an extreme case, two executables might have disjoint sets of observable behaviours, and yet both be correct with respect to a source C11 or C++11 program. To establish correctness, comparing the final checksum of the different binaries is not enough: we must ensure that all the behaviours of a compiled executable are allowed by the semantics of the source program. The Csmith experience suggests that large program sizes (∼80KB) are needed to maximise the chance of hitting corner cases of the optimizers; at the same time they must exhibit subtle interaction patterns (often unexpected, as in the example above) while being well defined (in particular data-race free). Capturing the set of all the behaviours of such large concurrent programs is tricky as it can depend on rare interactions between threads; computing all the behaviours allowed by the C11/C++11 semantics is even harder.

Despite this, we show that differential random testing can be used successfully for hunting concurrency compiler bugs, based on two observations. First, C and C++ compilers must support separate compilation and the concurrency model allows any function to be spawned as an independent thread. As a consequence compilers must always assume that the *sequential* code they are optimizing can be run in an *arbitrary concurrent context*, subject only to the constraint that the whole program is well defined (race-free on non atomic accesses, etc.), and can only apply optimizations that are sound with respect to the concurrency model. Second, we have shown in Chapter 4 that it is possible to characterise which optimizations are correct in C11 by observing how they eliminate, reorder, or introduce memory accesses in the traces of the sequential code with respect to a reference trace. We use in this chapter the model defined by the standard: $(\mathsf{ConsRFna}, \mathsf{SCorig}, \mathsf{RSorig}, \mathsf{STorig})$. Combined, these two remarks imply that testing the correctness of compilation of concurrent code can be reduced to validating the traces generated by running optimized sequential code against a reference (unoptimized) trace for the same code.

We illustrate this idea with program `foo.c` from Figure 5.1. Traces only report accesses to global (potentially shared) memory locations: optimizations affecting only the thread-local state cannot induce concurrency compiler bugs. On the left below, the reference trace for `th_1` initialises `x` and `y` and loads the value `1` from `x`:

```
                    Init  x 1
Init  x 1           Init  y 0
Init  y 0           Load  x 1
Load  x 1           Load  y 0
                    Store y 0
```

On the right above, the trace of the `gcc -O2` generated code performs an extra load and store to `y` and, since arbitrary store introduction is provably incorrect in the C11/C++11 concurrency model we can detect that a miscompilation happened. Figure 5.2 shows another example, of a randomly generated C function together with its reference trace and an optimized trace. In this case it is possible to match the reference trace (on the left) against the optimized trace (on the right) by a series of sound eliminations and reordering of actions.

5.2 Compiler Testing

Building on the theory of the previous section, we designed and implemented a bug-hunting tool called `cmmtest`. The tool performs random testing of C and C++ compilers, implementing a variant of Eide and Regehr's *access summary testing* [ER08]. A test case is any well defined, sequential C program; for each test case, `cmmtest`:

1. compiles the program using the compiler and compiler optimizations that are being tested;

2. runs the compiled program in an instrumented execution environment that logs all memory accesses to global variables and synchronisations;

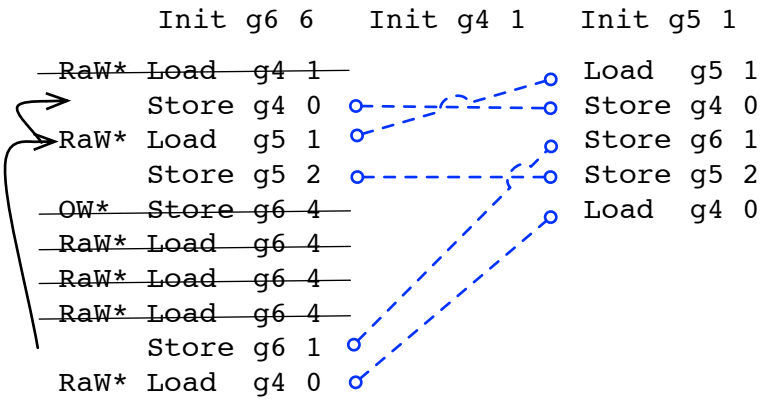
3. compares the recorded trace with a reference trace for the same program, checking if the recorded trace can be obtained from the reference trace by valid eliminations, reorderings and introductions.

Test-case generation We rely on a small extension of Csmith [YCER11] to generate random programs that cover a large subset of C while avoiding undefined and unspecified behaviours. We added mutex variables (as defined in `pthread.h`) and system calls to `pthread_mutex_lock` and `pthread_mutex_unlock`. Enforcing balancing of calls to lock and unlock along all possible execution paths of the generated test-cases is difficult, so mutex variables are declared with the attribute `PTHREAD_MUTEX_RECURSIVE`. We also added atomic variables and atomic accesses to atomic variables, labelled with a memory order attribute. For atomic variables we support both the C and the C++ syntax, the former not yet being widely supported by today's compilers. Due to limitations of our tracing infrastructure, for now we instruct Csmith to not generate programs with unions or consts.

```
const unsigned int g3 = 0UL;
long long g4 = 0x1;
int g6 = 6L;
unsigned int g5 = 1UL;

void f(){
  int *l8 = &g6;
  int l36 = 0x5E9D070FL;
  unsigned int l107 = 0xAA37C3ACL;
  g4 &= g3;
  g5++;
  int *l102 = &l36;
  for (g6 = 4; g6 < (-3); g6 += 1);
  l102 = &g6;
  *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

This randomly generated function generates the following traces if compiled with `gcc -O0` and `gcc -O2`.



All the events in the optimized trace can be matched with events in the reference trace by performing valid eliminations and reorderings of events. Eliminated events are ~~struck-off~~ while the reorderings are represented by the arrows.

Figure 5.2: successful matching of reference and optimized traces

Compilation Compilation is straightforward provided that the compiler does not attempt to perform whole-program optimizations. With whole-program optimization, the compiler can deduce that some functions will never run in a concurrent context and perform more aggressive optimizations, such as eliminating all memory accesses after the last I/O or synchronisation. It is very hard to determine precise conditions under which these aggressive whole-program optimizations remain sound; for instance Ševčík's criteria are overly conservative and result in false positives when testing Intel's `icc -O3`. Neither `gcc` nor `clang` perform whole-program optimizations at the typical optimization levels `-O2` or `-O3`.

Test case execution and tracing The goal of this phase is to record an execution trace for the compiled object file, that is the linearly ordered sets of actions it performs. For the x86_64 architecture we implemented a tracing framework by binary instrumentation using the Pin tool [LCM+05]. Our Pin application intercepts all memory reads and writes performed by the program and for each records the address accessed, the value read or written, and the size. In addition it also traces calls to `pthread_mutex_lock` and `pthread_mutex_unlock`, recording the address of the mutex. With the exception of access size information, entries in the execution trace correspond to the actions of Chapter 4. Access size information is needed to analyse optimizations that merge adjacent accesses, discussed below.

The "raw" trace depends on the actual addresses where the global variables have been allocated, which is impractical for our purposes. The trace is thus processed with additional information obtained from the ELF object file (directly via `objdump` or by parsing the debugging information). The addresses of the accesses are mapped to variable names and similarly for values that refer to addresses of variables; in doing so we also recover array and struct access information. For example, if the global variable `int g[10]` is allocated at `0x1000`, the raw action `Store 0x1008 0x1004 4` is mapped to the action `Store g[2] &g[1] 4`. After this analysis, execution traces are independent of the actual allocation addresses of the global variables. Finally, information about the initialisation of the global variables is added to the trace with the `Init` tag.

Irrelevant reads The Pin application also computes some data-flow information about the execution, recording all the store actions which *depend* on each read. This is done by tracking the flow of each read value across registers and thread-local memory. A dependency is reported if either the value read is used to compute the value written or used to determine the control that leads to a later write or synchronisation. With this information we reconstruct an approximation of the set of *irrelevant reads* of the execution: all reads whose returned value is never used to perform a write (or synchronisation) are labelled as irrelevant.

In addition, we use a second algorithm to identify irrelevant reads following their characterisation on the opsems: the program is replayed and its trace recorded again but binary instrumentation injects a different value for the read action being tested. The read is irrelevant if, despite the different value read, the rest of the trace is unchanged. This approach is slower but accurately identifies irrelevant reads inserted by the optimizer (for instance when reorganising chains of conditions on global variables).

Reference trace In the absence of a reference interpreter for C11, we reuse our tracing infrastructure to generate a trace of the program compiled at `-O0` and use this as the

reference trace. By doing so our tool might miss potential front-end concurrency bugs but there is no obvious alternative.

Trace matching Trace matching is performed in two phases. Initially, eliminable actions in the reference trace are identified and labelled as such. The labelling algorithm linearly scans the trace, recording the last action performed at any location and whether release/acquire actions have been encountered since; using this information each action is analysed and labelled following the definition of eliminable actions of Section 4.1.1. Irrelevant reads reported by the tracing infrastructure are labelled as such. To get an intuition for the occurrences of eliminable actions in program traces, out of 200 functions generated by Csmith with `-expr_complexity 3`, the average trace has 384 actions (max 15511) of which 280 (max 14096) are eliminable, distributed as follows:

IR	RaW	RaR	OW	WaR	WaW
8 (94)	94 (1965)	95 (10340)	75 (1232)	5 (305)	1 (310)

Additional complexity in labelling eliminable actions is due to the fact that a compiler performs many passes over the program and some actions may become eliminable only once other actions have been eliminated. Consider for instance this sequence of optimizations from the example in Figure 5.2:

```
Store g6 4
Load  g6 4
Load  g6 4  --RaW-->  Store g6 4  --OW-->  Store g6 1
Load  g6 4            Store g6 1
Store g6 1
```

In the original trace, the first store cannot be labelled as OW due to the intervening loads. However, if the optimizer first removes these loads (which is correct since they are RaW), it can subsequently remove the first store (which becomes an OW). Our analysis thus keeps track of the critical pairs between eliminations and can label actions eliminable under the assumption that other actions are themselves eliminated. Irrelevant and eliminable reads are also labelled in the optimized trace, to account for potential introductions.

Before describing the matching algorithm, we must account for one extra optimization family we omitted in the previous section: *merging of adjacent accesses*. Consider the following loop updating the global array `char g[10]`:

```
for (int l=0; l<10; l++) g[l] = 1;
```

The reference trace performs ten writes of one byte to the array `g`. The object code generated by `gcc -O3` performs only two memory accesses: `Store g 101010101010101 8` and `Store g[8] 101 2`. This optimization is sound under hypothesis similar to those of eliminations: there must not be intervined accesses or release/acquire pairs between the merged accesses; additionally the size of the merged access must be a power of two and the merged store must be aligned. Since the C11/C++11 memory model, as formalised by Batty et al., is agnostic to the size of accesses, we could not formally prove the soundness of this optimization.

The matching algorithm takes two annotated traces and scans them linearly, comparing one annotated action of the reference trace and one of the optimized trace at a time. It performs a depth-first search exploring at each step the following options:

- if the reference and optimized actions are equal, then consider them as matched and move to the next actions in the traces;
- if the reference action is eliminable, delete it from the reference trace and match the next reference action; similarly if the optimized action is eliminable. If the action is eliminable under some condition (for example an Overwritten Write (OW) is only eliminable if there is a later store to the same variable that is not eliminated, and any intervening read is eliminated), that condition is enforced at this point;
- if the optimized action can be matched by reordering actions in the reference trace, reorder the reference trace and match again.

The algorithm succeeds if it reaches a state where all the actions in the two input traces are either deleted or matched.

To deal with mergeable and splittable accesses (and improve speed), we split the traces by byte before running this algorithm. Each split traces has all the accesses to a single byte, plus all atomic operations (to detect illegal reorderings). This is correct because reordering non atomic accesses is always valid.

Some of these options are very expensive to explore (for instance when combinations of reordering, merging and eliminations must be considered), and we guide the depth-first search with a critical heuristic to decide the order in which the tree must be explored. The heuristic is dependent on the compiler being tested. The current implementation can match in a few minutes `gcc` traces of up to a few hundreds of actions on commodity hardware; these traces are well beyond manual analysis.

Outcome During tracing, `cmmtest` records one "execution" of a C or C++ program; using the terminology of previous sections, it should observe a witness for an opsem of the program. Since `cmmtest` traces sequential deterministic code in an empty concurrent environment, all reads of the opsem always return the last (in *sb* order) value written by the same thread to the same location. The witness is thus trivial: reads-from derives from *sb* while *mo* and are both included in *sb*. Note that the tool traces an execution of the generated assembler and as such it records a linearisation of the opsem's *sb* and not the original *sb*. In practice, our version of Csmith never generates programs with atomic accesses not related by *sb*, and we have not observed any false positive introduced by the linearisation of *sb* between non atomic accesses while using `cmmtest`.

The `cmmtest` tool compares two opsems and returns true if the optimized opsem can be obtained from the reference opsem by a sequence of sound eliminations/reorderings/introductions, and false otherwise. If `cmmtest` returns *true* then we deduce that this opsem (that is, this execution path of the program) has been compiled correctly, even if we cannot conclude that the whole program has been compiled correctly (which would require exploring all the opsems of the opsemset, or, equivalently, all the execution paths of the program). Because we are limited to checking one opsem, `cmmtest` is not *sound*: it will never discover wrong transformations such as:

```
r = a.load();
r2 = a.load();
```

by

```
r = a.load();
```

```
r2 = a.load();
if (r != r2)
  printf("this will never be seen in an empty context!");
```

More interestingly, `cmmtest` is not *complete*: whenever it returns *false*, then either the code has been miscompiled, or an exotic optimization has been applied (since our theory of sound optimizations is not complete). In this case we perform test-case reduction using CReduce [RCC+12], and manually inspect the assembler. Reduced test-cases have short traces (typically less than 20 events) and it is immediate to build discriminating contexts and produce bug reports. Currently the tool reports no false positives for `gcc` on any of the many thousands of test-cases we have tried without structs and atomics; we are getting closer to a similar result for arbitrary programs (with the restriction of a single atomic access per expression).

In practice, the tool is useful despite being neither sound nor complete in theory; the presence of both false negatives and false positives is acceptable as long as they are rare enough.

Stability against the HW memory model Compiled programs are executed on shared-memory multiprocessors which may expose behaviours that arise from hardware optimizations. Reference mappings of atomic instructions to x86 and ARM/POWER architectures have been proved correct [BOS+11, BMO+12]: these mappings insert all the necessary assembly synchronisation instructions to prevent hardware optimizations from breaking the C11/C++11 semantics. On x86_64, `cmmtest` additionally traces memory fences and locked instructions, and under the hypothesis that the reference trace is obtained by applying a correct mapping (e.g., by putting fence instructions after all sequentially consistent atomic writes), then `cmmtest` additionally ensures that the optimizer does not make hardware behaviours observable (for instance by moving a write after a fence).

Another interesting point is that despite tracing assembly code, we only need a theory of safe optimization for the C/C++11 memory model, and not for the x86 or ARM ones. It is indeed the responsibility of the compiler to respect the C11 model, regardless of the platform. So if the processor was to reorder instructions in an incorrect manner, it would still be a compiler bug, as the compiler should have put more fences to prevent such behavior.

5.3 Impact on compiler development

Concurrency compiler bugs While developing `cmmtest` and tuning the matching heuristic, we tested the latest svn version of the 4.7 and 4.8 branches of the `gcc` compiler. During these months we reported several concurrency bugs (including bugs no. 52558, 54149, 54900, and 54906 in the `gcc` bugzilla), which have all been promptly fixed by the `gcc` developers. In one case the bug report highlights an obscure corner case of the `gcc` optimizer, as shown by a discussion on the `gcc-patches` mailing list;[1] even though the reported test-case has been fixed, the bug has been left open until a general solution is found. In all cases the bugs were wrongly introduced writes, speculated by the LIM or IFCVT (if-conversion) phases, similar to the example in Figure 5.1. These bugs do not

[1] `http://gcc.gnu.org/ml/gcc-patches/2012-10/msg01411.html`

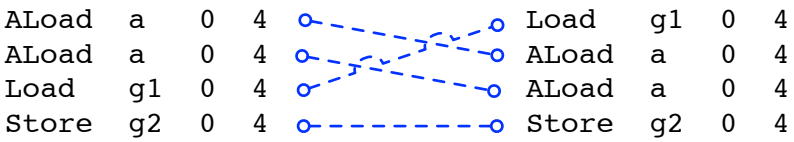
only break the C11/C++11 memory model, but also the Posix DRF-guarantee which is assumed by most concurrent software written in C and C++. The corresponding patches are activated via the `--param allow-store-data-races=0` flag for versions of GCC until 4.9 (it became default in 4.10) All these are *silent wrong-code bugs* for which the compiler issues no warning.

Unexpected behaviours Each compiler has its own set of internal invariants. If we tune the matching algorithm of `cmmtest` to check for compiler invariants rather than for the most permissive sound optimizations, it is possible to catch unexpected compiler behaviours. For instance, `gcc` 4.8 forbids all reorderings of a memory access with an atomic one. We baked this invariant into `cmmtest` and in less than two hours of testing on an 8-core machine we found the following testcase:

```
atomic_uint a; int32_t g1, g2;

int main (int, char *[]) {
  a.load () & a.load ();
  g2 = g1 != 0;
}
```

whose traces for the function `main` compiled with `gcc 4.8.0 20120627 (experimental)` at optimization levels `-O0` and `-O2` (or `-O3`) are:



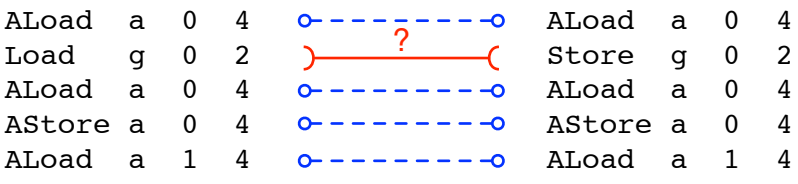
As we can observe, the optimizer moved the load of `g1` before the two atomic SC loads. Even though this reordering is not observable by a non racy context (see Section 4.2.2) , this unexpected compiler behaviour was fixed nevertheless (`rr190941`).

Interestingly, `cmmtest` found one unexpected compiler behaviour whose legitimacy is arguable. Consider the program below:

```
atomic_int a; uint16_t g;

void func_1 () {
  for (; a.load () <= 0; a.store (a.load () + 1))
    for (; g; g--);
}
```

Traces for the `func_1` function, compiled with `gcc 4.8.0 20120911 (experimental)` at optimization levels `-O0` and `-O2` (or `-O3`), are:



The optimizer here replaced the inner loop with a single write:

```
for (; g; g--);     ⟶     g = 0;
```

thus substituting a read access with a write access in the execution path where `g` already contains `0`. The introduced store is *idempotent*, as it rewrites the value that is already stored in memory. Since in the unoptimized trace there is already a load at the same location, this extra write cannot be observed by a non racy context.Strictly speaking, this is not a compiler bug. However, whether this should be allowed or not is subject to debate. In a world with hardware or software race detection it might fire a false positive. It can also cause a fault if the page is write-protected. Also, possibly a bigger issue, the introduced write can introduce cache contention where there should not be any, potentially resulting in an unexpected performance loss.

Chapter 6

LibKPN: A scheduler for Kahn Process Networks using C11 atomics

In the previous chapters we have studied C11 from the point of view of compiler writers. We will now investigate how C11 atomics can be used by programmers to write efficient yet portable concurrent code through an example.

That example is LibKPN, a dynamic scheduler for a particular class of deterministic concurrent programs called Kahn Process Networks (KPN). After explaining what they are and why scheduling them efficiently is challenging, we will present a naive algorithm for doing so and progressively refine it to account for the C11 memory model. We later prove that this algorithm is free from undefined behaviour and does not deadlock on valid KPNs, showing that despite all of its flaws the C11 model is usable today.

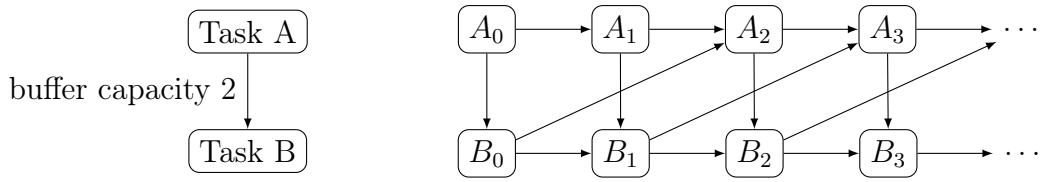
6.1 Scheduling of Kahn Process Networks

Kahn Process Networks (KPN) [Kah74] are a widely studied paradigm of parallel programming. A KPN is composed of two kinds of objects: processes and queues (also called *tasks* and *channels*). All queues are single producer single consumer (SPSC), first-in first-out (FIFO), and typically bounded. Each process executes a sequential program, that can only communicate with the other processes through two operations: pushing a value on a queue, and popping a value from a queue. An important feature of KPN is that pop is blocking, i.e. if there is no value to read, it waits until one becomes available. In the case of bounded queues, push is also blocking, i.e. if there is no space to push a value, the task also waits until space becomes available.

KPNs are the basis of the Lustre language [CPHP87], and inspired more recently Openstream [PC13], an extension of OpenMP with streaming.

An interesting property of this model is that it is only possible to write deterministic programs in it, i.e. programs whose outputs are always the same when given the same inputs. The model does not allow writing all deterministic programs. For example it is impossible to implement the *parallel or* operation. This is the operation defined by the following:

- it has two inputs and one output;
- it pushes true on its output as soon as true is available on one of the two;
- it pushes false once false is available on both;



(a) A basic KPN with two tasks

(b) The same KPN as a network of task activations with dependencies

Figure 6.1: An example of decomposing KPNs into task activations

The reason it cannot be implemented as a KPN is that it must wait on both of its inputs simultaneously, and KPNs only offer a blocking pop operation that looks at one input. We will see in Section 6.2.4 how to deal with a richer model that can implement parallel or.

6.1.1 Graphs of dependencies between task activations

An alternative way of thinking about parallel deterministic programs is based on *task activations*[1] and explicit dependencies between these. In this model, each task activation has a list of other task activations it depends on and can only start once they are finished.

These two different models turn out to be equivalent [Lê16].

A KPN can be represented by splitting each process (that we call task in the rest of this thesis) into task activations, separated at every push and pop. These task activations are then related by a chain of dependencies to enforce the original program order inside tasks. Finally, extra dependencies are added from every push to the pop that reads from it and from every pop to the push that later stores to the memory cell made available by that pop. Consider for example the Figure 6.1. On the left, it shows a basic KPN: two tasks A and B, connected by a buffer of size 2. A repeatedly pushes into the buffer, and B repeatedly pulls from it. On the right is the equivalent representation as task activations connected by dependencies. The arrows there represent the dependencies (no longer the buffer).

Conversely, we can represent each task activation by a KPN process, and each dependency by a single-use queue, that is pushed into when the dependency is satisfied, and popped from at the start of the task waiting for it. This representation is somewhat inefficient, but it can be used as a first step when translating existing parallel implementations to LibKPN. OpenMP in particular is best understood in the tasks activations and dependencies model. Efficiency can then be recovered by aggregating task activations along a dependency chain in a single task, and aggregating dependencies between parallel tasks into full-blown channels.

6.1.2 Scheduling: why naive approaches fail

All tasks can be classified in three groups:

- currently running on a core;
- ready to run;

[1] Task activations are more often called tasks in the literature, but we already use the word task for kahn processes

- waiting on a dependency, i.e. trying to push to a full queue or pop from an empty queue;

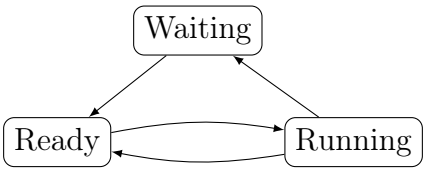


Figure 6.2: The three logical states a task can be in

There has already been a lot of research on how to schedule lightweight threads on the cores of modern multicore processors. We can reuse these results to deal with the transitions between the ready and running states in Figure 6.2. More specifically we use a work-stealing algorithm for this purpose (see next section for details).

What we focus on is the other transitions: tracking which task is ready to run, and which one is still waiting on a dependency.

6.1.2.1 Polling tasks for readiness

A simple approach would be to consider all tasks as ready to run, and check the availability of their dependencies when they start running. This amounts to polling not-running tasks regularly to find those actually ready to run. Unfortunately, this approach does not work for two reasons, and we must explicitly track separately tasks ready to run and those waiting on a dependency.

The first issue is that this approach limits us to schedulers that exhibit some degree of fairness. Consider a trivial example with only one core and two tasks A and B, such that A is ready to run and B depends on A. If the scheduler is unfair and constantly choose to run B, the system will not make progress. This is especially an issue for schedulers based on work-stealing, that offer little overhead and good locality, but have no fairness by default.

The second issue is that even with a perfectly fair scheduler, the performance can be arbitrarily degraded. Consider a set of n tasks $A_0, A_1, \dots A_n$ where $A_{(i+1)}$ depends on A_i. If a round-robin scheduler (perfectly fair) tries to execute this set without taking into account the dependencies, it may run the tasks in reverse order, resulting in $n-1$ useless task activations before A_0 can make progress, then $n-2$ before A_1 makes progress and so on. As n is not bounded, this can represent an arbitrarily large slow-down.

6.1.2.2 Naive cooperative algorithm

A simple way to track the readiness of tasks is to add two bits to each buffer. Whenever a task tries to pop from an empty buffer, it sets the first bit, and whenever it tries to push from a full buffer, it sets the second bit. In both cases, it then yields control back to the scheduler. We say that this task is *suspended*. Whenever a task successfully pushes or pops from a buffer, it checks the corresponding bit and if it is set, unsets it and informs the scheduler that the task on the other side is now ready to run. We say that this task is *woken up*.

This algorithm fails because of a race between the moment where a task sets the bit, and the moment where the opposing task tests it. The next section will describe increasingly efficient ways of dealing with this race.

6.2 Algorithms for passive waiting

In this section, we will see the pseudocode for the three different algorithms we developed, along with informal justifications of their correctness. The formal proofs are in the next section.

The first algorithm solves the race explained previously by adding a second check in suspend. We call it algorithm F (for Fence) in the figures because in C11 it requires a sequentially consistent fence in push and pop. It is presented in Section 6.2.1.

The second algorithm uses an optimistic approach: it lets tasks get suspended wrongly, and makes sure they are awoken later when one of their neighbors suspends. We call it algorithm S for Scan, because it requires a scan of all neighbors of a task during suspension. It is presented in Section 6.2.2.

We then show a way of getting the performance benefits of both approaches with a hybrid algorithm, called H in the figures. It is presented in Section 6.2.3.

Finally, we discover in Section 6.2.4 that it is possible to adapt this last algorithm to implement a richer model than KPNs.

Our code manipulate three kinds of data structures:

Tasks Each task has a pointer to the function it represents. It also has a pointer to an array of all the queues it is connected to. Finally, it has a pointer to the worker thread it is executing on.

Work-stealing deques There is one such worker thread for each core. Each worker thread has a work-stealing deque filled with task pointers. Such a deque offers three operations: take and give that can be used by a thread to act on its own deque; and steal which can be used by other threads to get tasks to execute when their own deque is empty. We use for this purpose the Chase-Lev deque [CL05], which has been adapted for C11 and proven correct in this context in [LPCZN13]. The three functions from this library are prefixed by `WS`.

Queue Each queue (channel between tasks) has two bits, one for the producer to indicate it is waiting for empty space to push, and one for the consumer to indicate it is waiting for a value to read. It also has a ring buffer to store the data. For this part we use a C11 library that is described and proven correct in [LGCP13]. The functions from this library are prefixed by `RB`. Finally it has pointers to both its producer and its consumer.

We present the code to all three algorithms in Figures 6.3 and 6.4. `push` and `pop` take as argument the task executing them, the queue to operate on, and the data to write or the variable to store the result in. They can directly call `tryawaken` on a task and the bit in the queue noting whether that task is suspended. Finally, they can ask the scheduler to call `suspend` on a task on their behalf[2]. The function `suspend` also takes as argument the queue the task is suspending on, and an enum telling whether the task is suspending during a push or a pop.

6.2.1 Algorithm F: double-checking in `suspend`

As the case of popping from an empty buffer, and the case of pushing into a full one are symmetric, we will only treat the first case in what follows. In essence, what we build can

[2]We explain why `push` and `pop` cannot call `suspend` directly themselves in Section 6.2.5.2.

```
push(t, q, x) {
  if (RB_is_full(q->buf)) {
    yield_to_scheduler(t->worker, SUSPEND(t, q, OUT));
  }
  RB_push(q->buf, x);
  #if ALGORITHM_F
    fence(seq_cst);
  #endif
  tryawaken(&q->wait[IN], q->task[IN]);
}

pop(t, q, var) {
  if (RB_is_empty(q->buf)) {
    yield_to_scheduler(t->worker, SUSPEND(t, q, IN));
  }
  RB_pop(q->buf, var);
  #if ALGORITHM_F
    fence(seq_cst);
  #endif
  tryawaken(&q->wait[OUT], q->task[OUT]);
}
```

Figure 6.3: Code of `push` and `pop`

be seen as an SPSC semaphore; and each bounded channel has two: one for tracking the available space for push, and one for tracking the available data for pop.

We remove the race mentioned in the previous section by adding a second check after setting the bit; if that second check shows that another thread pushed into the buffer while we were setting the bit, then we try to unset the bit and cancel our suspension.

This introduces a new race as we can try to cancel our suspension at the same time that the other thread wakes us up. That new problem is solved by using an atomic exchange operation on the bit: no more than one thread can get the old value of 1.

The first load of `waitvar` in `tryawaken` is purely an optimization; doing the exchange operation unconditionally would also be correct, merely a bit slower.

Translation to the C11 model Until now, we designed the algorithm as if it was running on the SC model. As the code is racy, we must now account for the subtleties of the C11 model.

In particular, it is necessary to add sequentially consistent fences to both suspend and push/pop, to prevent a situation analogous to the SB litmus test, in which neither the load in `tryawaken`, nor the load in `suspend` see the stores in `push`/`suspend` (see Figure 6.5). This would result in the task never being awoken, leading to deadlock.

We have seen in Section 3.0.1 that SC fences have semantics that are weaker than expected. But it is not a problem here as we only need to prevent the SB-like pattern and the rules clearly work for this. For the SB pattern to have its anomalous result, one of the two writes would have to be before in the mo order the write of the initial value by SCFences3. This in turn is impossible by CohWW.

```
is_not_full_or_not_empty(q, d) {
    return (d == IN && !RB_is_empty(q->buf))
        || (d == OUT && !RB_is_full(q->buf));
}

suspend(t, q, d) {
  #if ALGORITHM_F || ALGORITHM_S
    store(&q->wait[d], 1, release);
  #elif ALGORITHM_H
    store(&q->wait[d], 1, relaxed);
    store(&t->status, 1, release);
  #endif
  fence(seq_cst);
  #if ALGORITHM_F
    if (is_not_full_or_not_empty(q, d))
      tryawaken(&q->wait[d], t);
  #elif ALGORITHM_S
    for (kind in {IN, OUT}) {
      for (p in t->queues[kind])
        tryawaken(&p->wait[kind], p->task[kind]);
    }
  #elif ALGORITHM_H
    s = load(&q->task[!d]->status, acquire);
    if (s && is_not_full_or_not_empty(q, d))
      tryawaken(&q->wait[d], t);
    for (kind in {IN, OUT}) {
      for (p in t->queues[kind])
        if (is_not_full_or_not_empty(p, kind))
          tryawaken(&p->wait[kind], p->task[kind]);
  #endif
}

tryawaken(waitvar, t) {
  if (load(waitvar, relaxed)) {
    #if ALGORITHM_F || ALGORITHM_S
      wv = exchange(waitvar, 0, acquire);
      if (wv)
        WS_give(workq[t->worker], t);
    #elif ALGORITHM_H
      s = exchange(&t->status, 0, acquire)
      if (s) {
        *waitvar = 0;
        WS_give(workq[t->worker], t);
      }
    #endif
  }
}
```

Figure 6.4: Code of `suspend` and `tryawaken` (relaxed code in *italics*)

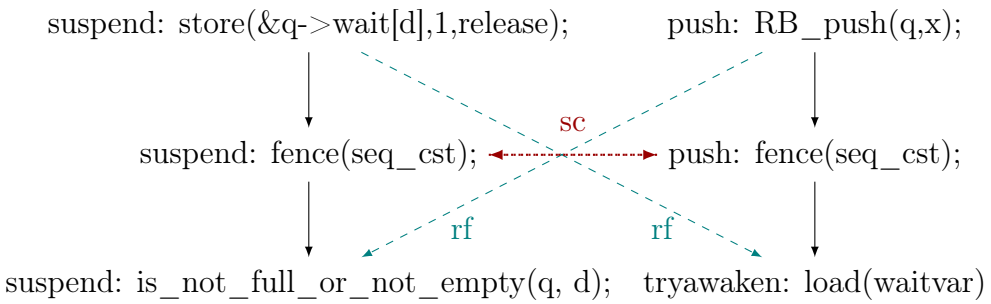


Figure 6.5: SB-like pattern in algorithm F: depending on the direction of the sc order, at least one of the rf edges is present.

The acquire exchange operation in `tryawaken` synchronises with the release store in `suspend`, to make sure that if that task is awakened, the new activation of it will see everything done by the previous activation, giving the illusion of sequentiality. At the same time, the first load in `tryawaken` can be relaxed, as there is no need to synchronise: the exchange operation will take care of it.

6.2.2 Algorithm S: removing fences on the critical path

The previous algorithm has a frustrating issue: it has a rather costly sequentially consistent fence in the middle of every push and pop. This is especially regrettable after having picked an underlying ring buffer data structure precisely for its low overhead and absence of full fence on the critical path.

Our first solution to this issue is the algorithm S (which stands for Scan), which enforces a weaker invariant[3]. Informally, the algorithm F enforces the invariant that no task is ever suspended on a buffer wrongly, i.e. waiting for data while the buffer is not empty. More formally: if the suspend operation succeeds, there is no push operation whose fence is sc-before the one in the suspend procedure that the pop operation could have read from. It turns out that a much weaker invariant is sufficient to prevent deadlocks: no task is ever wrongly suspended waiting on another task that was wrongly suspended after it.

This is achieved by having `suspend` unconditionally succeed, and do a scan of all other tasks potentially waiting on the one suspending, waking all of them up. Those that were wrongly suspended will be able to make progress; the rest will suspend again (potentially waking up more tasks, and so on). This cascade of wake-ups will eventually reach any wrongly suspended task, ensuring progress of the whole system.

6.2.3 Algorithm H: keeping the best of algorithms F and S

The algorithm S has very low overhead on most microbenchmarks. But it performs badly on a few. The reason is that it can spuriously wake-up and suspend again every task in the network every time a task must suspend. This motivates the search for a third algorithm that would have the low worst-case complexity of algorithm F, and the low overhead in the average case of algorithm S.

[3]Such reasoning based on invariants is hard to state formally in weak memory models, as there is no one ordering of memory operations that all threads can agree upon. It is provided here to give the intuition that led us to the algorithm, the formal proof is written in a different style, and is available in the next section.

This hybrid algorithm (called algorithm H in the pseudo-code fragments) relies on the following informal invariant: no task is ever suspended wrongly on a task that is also suspended. A deadlock would require a cycle of tasks suspended on each other. With this invariant, it is clear that such a cycle is impossible unless all the suspensions are justified, which would be a sign of an incorrectly specified KPN.

Its enforcement is made possible by moving the point of synchronisation from the buffers to the tasks themselves, each task having a status field that is set whenever it is suspended. When a task A suspends on a task B, A starts by reading this status bit from B. If it is not set, then it can suspend unconditionally: it will be B's responsibility to enforce the invariant. If it is set, then A must check again that it is correct to suspend. This second check is guaranteed to be accurate because the load of the status bit is acquire and the related store is release, so A sees everything that B did before suspending. Whenever a task successfully suspends, it then scans all of its neighbours to find if any are wrongly suspended on it. This last part is what lets B enforce the invariant when A does not see its status bit set.

To recapitulate, if a task A is about to suspend wrongly on a task B that is suspended or suspending, there are three possibilities:

- the task A will see the task B suspended when it loads its status bit. Since this load is acquire, and the setting of this bit uses the release attribute, everything done to this point by task B is visible to task A. A can now cancel its suspension as it notices the push that B made prior to suspending;
- or B will notice that A is suspending when doing its scan at the end of the `suspend` procedure, and wake it up;
- or both will happen, and they will both call `tryawaken` on A. This is not an issue, thanks to the `exchange` operation which will succeed for exactly one of them.

It is impossible for both to miss the other, because of the sequentially consistent fence in `suspend`; the code has the same shape as the SB litmus test.

Since cascade of wake-ups are not required in this algorithm, the scan does not have to wake-up tasks that are not wrongly suspended. As a result, it can be proved that there are no *spurious wake-ups*: any time a task is woken up, it can make progress (defined as executing outside of push and pop).

It is interesting to relate these three algorithms to the "Laws of Order" paper [AGH+11]. That paper shows that all concurrent operations that respect some conditions *must* use at least one sequentially consistent fence, or a read-modify-write operation. We do have such a fence in `push` and `pop` in Algorithm F, but not in the other two algorithms. The reason we can escape this requirement is that `push`/`pop` are not linearizable in algorithms S and H (we can have a `push` and a `pop` on the same queue that do not notice each other and have inconsistent views of the queue), and linearizability is one of the conditions established in the paper. On the other end, when we actually get some mutual exclusion (only one instance of a task can be woken up at a time), we do get costly synchronisation (some form of exchange or compare-and-exchange) in all three algorithms.

6.2.4 Going beyond KPNs: adding *select* in algorithm H

Since the memory location that is used for synchronisation is now in the task itself, it is safe to have different threads call `tryawaken` on the same task; only one will succeed thanks

```
select(t, qs, var) {
  while (true) {
    foreach (q in qs) {
      if (!RB_empty(q->buf)) {
        RB_pop(q->buf, var);
        tryawaken(q->tasks[OUT], q, OUT);
        return;
      }
    }
    yield_to_scheduler(t->worker, SUSPEND_SELECT(t, qs));
  }
}

suspend_select(t, qs) {
  t->select_qs = qs;
  foreach (q in qs) {
    store(&q->wait[IN], 1, relaxed);
  }
  store(&t->status, 1, release);
  fence(seq_cst);
  foreach (q in qs) {
    s = load(q->task[OUT]->status, acquire);
    if (s && !RB_empty(q->empty)) {
      tryawaken(t, q, IN);
      break;
    }
  }
  for (kind in {IN, OUT})
    for (p in t->queues[kind])
      if (is_not_full_or_not_empty(p, kind))
        tryawaken(p->task[kind], p, kind);
}

tryawaken(t, q, kind) {
  wv = load(&q->wait[kind], relaxed);
  if (wv) {
    s = exchange(&t->status, 0, acquire)
    if (!s)
      return; // cancel wake-up
    foreach (p in t->select_qs) {
      store(&p->wait[kind], 0, relaxed);
    }
    WS_give(workq[t->worker], t);
  }
}
```

Figure 6.6: Supporting `select` in algorithm H

to the atomic exchange operation. This allows us to implement the `select` operation, that pops from any buffer from a set that is non empty. The corresponding pseudo-code is presented in Figure 6.6. The presence of this operation means that we are now strictly more expressive than Kahn Process Networks. For example, we can now easily implement *parallel or*.

Preventing spurious wake-ups caused by select If we implement select as in Figure 6.6, we will lose the property that every task that is woken up can make progress. Consider the following sequence of events:

- task A does a select on B and C and suspends itself;
- task B reads the wait field of the queue to A (in a call to tryawaken);
- task C wakes up A;
- task A does a select on D and E and suspends itself;
- task B swaps the status field of A to 0, observes it was positive and wakes up A;
- task A is awoken spuriously: it has no data to read from D or E

It is easy to prevent this at the cost of turning the exchange operation in `try_awaken` into a compare-and-exchange operation, as shown in Figure 6.7. The trick is to keep in each task an `epoch` field that counts the number of times that task suspended itself, to store it into both the status field and the queue wait field, and to have `try_awaken` check the correspondance between the two. In the sequence of events above, the fifth step would then fail, as B would notice that the value it read in the wait field of the queue is stale. It must thus retry, to check whether A is still selecting on it or not (and since it is not, it will eventually notice the store of 0 into the wait field of the queue by C).

This situation, where an object goes from a first state (in this case suspended) to a second state (woken up) and back (to suspended) is a common problem called ABA [DPS10]. Using a counter to make the compare-and-swap fail in that case is a common solution where it is possible.

6.2.5 Implementation details

The code we have shown until now is idealised, without concerns for implementation details like memory layout or memory management. We will now deal with these issues and a few more.

6.2.5.1 Termination

The scheduler loop is shown in Figure 6.8. In it, we can see that termination is detected by a straightforward pair of counters: `num_spawned` which is incremented whenever a task is created and given to the scheduler, and `num_collected` which is incremented whenever a task is over. Care must be taken to load them in the order shown, otherwise the following could happen and cause some worker threads to exit too early:

- task A is the last to execute;
- some worker threads load `num_spawned` and sees the value n;

```
suspend_select(t, qs) {
  foreach (q in qs) {
    store(&q->wait[IN], t->epoch, relaxed);
  }
  store(&t->status, t->epoch, release);
  ++t->epoch;
  fence(seq_cst);
  foreach (q in qs) {
    s = load(q->task[OUT]->status, acquire);
    if (s && !RB_empty(q->empty)) {
      tryawaken(t, q, IN);
      break;
    }
  }
  for (kind in {IN, OUT})
    for (p in t->queues[kind])
      if (is_not_full_or_not_empty(p, kind))
        tryawaken(p->task[kind], p, kind);
}

tryawaken(t, q, kind) {
  while ((wv = load(&q->wait[kind], relaxed))) {
    s = CAS(&t->status, wv, 0, acquire, relaxed);
    if (!s)
      continue; // wv is stale, retry
    foreach (p in t->select_qs) {
      store(&p->wait[kind], 0, relaxed);
    }
    WS_give(workq[t->worker], t);
    return; // success
  }
}
```

Figure 6.7: Preventing spurious wake-ups in algorithm H with select

```
while (true) {
  task *tp = NULL;
  while (tp == NULL) {
    int collected = load (&num_collected, acquire);
    int spawned = load(&num_spawned, relaxed);
    if (collected == spawned) {
      exit(0);
    }
    tp = WS_try_pop();
    if (tp == NULL)
      tp = WS_try_steal();
  }
  result = run(tp);
  if (result == DONE) {
    suspend_done(t);
    fetch_and_add(&num_collected, 1, release);
  } else if (result == SUSPEND(t, q, d))
    suspend(t, q, d);
  else if (result == SUSPEND_SELECT(t, qs))
    suspend_select(t, qs);
}
```

Figure 6.8: Code of the scheduler loop

- task A creates a task B and gives it to one worker thread, incrementing `num_spawned`;
- task A ends, and `num_collected` is incremented;
- the worker threads load `num_collected` and sees the value n;
- they decide that all tasks are over, and exits while task B still has not finished.

In addition to incrementing `num_collected`, the end of a task leads to a call to `suspend_done` (shown in Figure 6.9). This function acts like `suspend` on a phantom queue. Its role is to make sure that no other task is wrongly suspended on that one. Since tasks cannot wrongly suspend in algorithm F, it is a no-op in that case.

6.2.5.2 Avoiding a race on stacks

Another point about the scheduler loop is that `suspend` is always called from the scheduler loop, and not directly from the code of `push`/`pop` (in Figure 6.3). This is absolutely necessary as other threads can wake up the task as soon as `suspend` stores to `q->wait` (in the case of the first two algorithms) or to `t->status` (in the case of the last algorithm). At this point, they may access the stack of the task, and if the task is still executing (for example because it is doing the scan in `suspend`) this will cause some data-race and stack corruption.

```
suspend_done(t) {
  #if ALGORITHM_H
    store(&t->status, 1, release);
    fence(seq_cst);
    for (kind in {IN, OUT}) {
      for (p in t->queues[kind])
        if (is_not_full_or_not_empty(p, kind))
          tryawaken(&p->wait[kind], p->task[kind]);
  #elif ALGORITHM_S
    fence(seq_cst);
    for (kind in {IN, OUT}) {
      for (p in t->queues[kind])
        tryawaken(&p->wait[kind], p->task[kind]);
    }
  #endif
}
```

Figure 6.9: Code of `suspend_done`

6.2.5.3 Storing wait fields together for better cache locality

Both algorithms H and S require tasks to be able to scan all the queues they are connected to, and do something when one of their two wait fields is set. This can be very costly for tasks connected to many queues if it involves a pointer dereference for each queue (and thus potentially a cache miss). Stores to these fields are on the other hand quite rare. So we move these fields from the queue structure to an array in the task structure. When the producer suspends, it must follow a pointer from the queue to the consumer task. On the other hand, the consumer during its scan can just iterate over the array looking for a non null cell.

This restricts expressiveness in one way: the number of queues that will be attached to a task must be known when allocating the task to correctly size the array. This does not seem to be a problem in our examples. If it were to become a problem, it would be possible to mitigate it, for example by using a linked list of arrays for tasks with a very large number of attached queues.

6.2.5.4 Memory management

There are several kinds of objects to keep on the heap and eventually reclaim.

Work-stealing deques This one is by far the easiest. As long as one scheduler loop is active, these are kept. The last scheduler loop to exit can reclaim them all. Reallocation when the deques are full is managed completely transparently by the library we are using.

Queues A queue can be accessed by exactly two tasks as it is single-producer single-consumer. So we use a simple atomic reference count starting at two. When a task closes the queue or finishes it decrements the reference count. When it hits 0, the queue is freed.

Tasks Tasks are much more difficult to manage, as they can be accessed concurrently by any other task that shares a queue with them, in addition to itself. In particular, in algorithm H their status field can be read.

This is solved again with a reference count, initialized at one and incremented by every queue the task is connected to. When a queue is reclaimed, it decrements the reference count of both sides. When the task finishes, it also decrements its own reference count. Whoever brings the reference count to 0 can safely free the task.

Vector of wait fields As explained in Section 6.2.5.3, the wait fields of all queues connected to a given task are stored in an array that is owned by that task. This array can be safely reclaimed at the same time that the task structure itself is.

6.2.5.5 Batching queue operations

The library we use for the queues offers batch operations: pushing or popping n values at once. We expose these operations to the programmer, but they can introduce another kind of spurious wake-ups:

- task A tries to pop 3 values from a queue q connecting it to task B. q is empty, so it fails and A suspends.
- task B pushes one value to q, sees that A is suspended and wakes it up.
- task A tries again, but there is only one value in q, so it suspends again without having made progress.

This can be solved by storing in the `q->wait` field the number of elements missing, and only waking up a suspended task while it is reached. Since we already need to store the epoch counter in this field (see Section 6.2.4), we can store one value in the high bits, and the other in the low bits.

6.3 Proofs of correctness

6.3.1 Outline of the difficulties and methods

A common approach to proving concurrent algorithms is linearizability. However, because it relies implicitly on a total order among operations it can only be directly used in a sequentially consistent model and fails to capture the specific guarantees provided by operations in a weak memory model. For example in our case, a successful pop happens-after a successful push, but there is no ($\prec_{\mathsf{sc}}$) edge in-between. There has been an attempt to build a similar approach for modularly proving programs in C11 [BDG13], but it becomes non-modular and rather complex for programs that use relaxed atomics (such as LibKPN).

Another approach is to use a program logic. Sadly, none of those that have been proposed for C11 [TVD14, VN13] cover the entire subset of C11 features we use (notably sequentially consistent fences + relaxed atomics). So we went instead for a proof that is specific to this program, and not modular, but can deal with all the parts of C11.

The most important thing to show is that tasks cannot be duplicated; in other words a task gets suspended between every pair of (adjacent) wake-ups. If a task was awoken

twice in a row, the two resulting task activations would race on every access, which would make the whole program undefined.

Since the existence of a duplicated task can lead to more duplicated tasks, this proof needs to proceed by induction: no task is duplicated at the beginning of time, and every transition of the system preserves this property. However, the very existence of the $\prec_{\mathsf{hb}}$ order over the transitions of the system crucially depends on the absence of duplicated tasks and races, making it unavailable as the order to do the induction over. We solve this by building a custom $\prec_{\mathsf{to}}$ order that is clearly well-founded, and only showing it to be a subset of $\prec_{\mathsf{hb}}$ at the very end of the proof.

This proof that tasks are never duplicated and have the illusion of sequentiality is shared between the three algorithms. They differ in the proof of progress that follows (Section 6.3.3).

6.3.2 Absence of races

Definition 17. *Any task can be in one of 7 different* states*:*

- U *uninitialised*
- N *new*
- P *posted*
- E *executing*
- S *suspended*
- W *waking-up*
- F *finished*

Definition 18. *Some instructions in the code are called* transition points*.*
Each of these points is associated to a state that it transitions the task to.
Here is the list of these points:

spawn *N: the increment of* `num_spawned` *when a task is created*

finish *F: the increment of* `num_collected` *in the scheduler loop when a task ends*

post *P: the call to* `WS_give` *by the creator of a task*

take *E: the call to* `WS_take` *in the scheduler loop*

steal *E: the call to* `WS_steal` *in the scheduler loop*

suspend *S: the first release store in* `suspend` *or* `suspend_select` *(so a store to* `t->status` *for algorithm H, and to* `q->wait` *for the other two)*

wakeup *W: the Read-Modify-Write operation in* `tryawaken`*, when it reads a positive value*

repost *P: the call to* `WS_give` *in tryawaken*

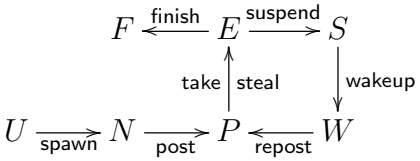
Definition 19. *We define* $(\prec_{\mathsf{to}})$ *as an order on transition points that is compatible with* $(\prec_{\mathsf{hb}})$ *(i.e.* $(\prec_{\mathsf{to}} \cup \prec_{\mathsf{hb}})^+$ *is acyclic) and whose projection on the transition points of any given task* T *is a total order.*
$(\prec_{\mathsf{to}})$ *exists by the order-extension principle (any partial order can be extended into a total one).*
When before *or* after *are used without precision in this document, they are meant to use* $(\prec_{\mathsf{to}})$*.*

Definition 20. *A task T is said to be in a state A if the last transition by $(\prec_{\mathsf{to}})$ that is associated with T leads to A (is associated with A).*
Before (by $(\prec_{\mathsf{to}})$) its first transition, a task is said to be in the state U.

Definition 21. *We say that a transition t of a task T respects* H *if all the following statements are true:*

- $\forall t_2 \prec_{\mathsf{to}} t, t_2 \prec_{\mathsf{hb}} t$.
- *If t is spawn, then T was in U just before*
- *If t is finish, then T was in E just before*
- *If t is post, then T was in N just before*
- *If t is take, then T was in P just before*
- *If t is steal, then T was in P just before*
- *If t is suspend, then T was in E just before*
- *If t is wakeup, then T was in S just before*
- *If t is repost, then T was in W just before*
- *If t is finish, suspend or repost then it is $(\prec_{\mathsf{sb}})$-after the $(\prec_{\mathsf{to}})$-last transition of T*
- *If t is take, steal or wakeup, then it reads from the latest transition to P*

H is the induction hypothesis used in the proof. Most importantly, it says that tasks states change according to the following automaton:



Lemma 2. $(\prec_{\mathsf{to}} \cup \prec_{\mathsf{hb}})^+$ *is a well-founded order.*

Proof. The relation is trivially transitive, as it is a transitive closure. It is also trivially acyclic by definition of $(\prec_{\mathsf{to}})$.

Let us show that it is well-founded.

Let there be an infinite decreasing sequence. We project it on every thread. Since there is only a finite number of threads, at least one projection is also infinite. $(\prec_{\mathsf{sb}}) \subset (\prec_{\mathsf{hb}})$, so every element of this sequence is either incomparable or smaller than every previous element of the sequence.

Statements in C are always comparable according to $(\prec_{\mathsf{sb}})$, while operands within an expression may not be, because of unspecified evaluation order. Since expressions in the C program are finite, we assume that every event is incomparable with only a finite number of elements; i.e., that every operator chain has a finite arity.

Then we can quotient by the equivalence relation ("is incomparable with") $\cup (=)$, and get an infinite, decreasing sequence in $(\prec_{\mathsf{sb}})$, which cannot exist. □

Lemma 3. *Every transition respects* H.

Proof. We proceed by induction over $(\prec_{\mathsf{to}} \cup \prec_{\mathsf{hb}})^+$ (which is a well-founded order by the previous lemma) with the induction hypothesis H. Let t be a transition of a task T, such that $\forall t_2 \prec_{(\mathsf{to} \cup \mathsf{hb})^+} t$, t_2 respects H, we must prove that t respect H.

By case analysis over t:

- If t is a *spawn* transition.
  If there were any transition $(\prec_{\mathsf{to}})$before t, the first such transition would be a spawn transition t_2 by H. The spawn transition always happens on a task just $(\prec_{\mathsf{sb}})$-after it is allocated. So the task would have been allocated both just before t and just before t_2. This is a contradiction, so t must be the first transition T. So t respects H trivially.

- If t is a *finish* or *suspend* transition.
  From the code of the scheduler loop, it is clear that there is a take or steal transition t_2 of T that is $(\prec_{\mathsf{sb}})$-before t, with no other transition $(\prec_{\mathsf{sb}})$-between them. By contradiction: let's assume that there is another transition t_2 of T such that $t_E \prec_{\mathsf{to}} t_2 \prec_{\mathsf{to}} t$. We pick the $(\prec_{\mathsf{to}})$-first such t_2. We can apply H to t_2, and T is in E after t_E, so t_2 is necessarily another finish or suspend transition. Again by H, t_2 must be immediately $(\prec_{\mathsf{sb}})$-after the previous transition, that is t_E. But that is absurd as t is the transition of T immediately $(\prec_{\mathsf{sb}})$-after t_E. This transition is unique, because we kept $(\prec_{\mathsf{sb}})$ total on the actions of each thread, by avoiding having memory accesses either as arguments of functions or as operands to operators (the two main ways for $(\prec_{\mathsf{sb}})$ to split).

- If t is a *post* transition.
  We assume that `starnk_post()` is only called once by task, and that it is called $(\prec_{\mathsf{sb}})$-after the task is spawned.
  So there is a $t_s \prec_{\mathsf{hb}} t$ spawn transition of T. By H, the only way to leave the state N is by a post transition. But we just assumed that t is the only such transition. So there cannot be any transition of T between t_s and t, and t respects H.

- If t is a *take* or *steal* transition.
  t gets the task T from a work-stealing deque, so by property of the work-stealing deque, there is t_P a post or repost transition of T to P with $t_P \prec_{\mathsf{hb}} t$, and t_P justifying t. So $t_P \prec_{\mathsf{to}} t$. By contradiction: assume that there is another transition t_2 $(\prec_{\mathsf{to}})$-between t_P and t. We pick the first such t_2. H applies to t_2, so t_2 is a take or steal transition. By property of the work-stealing deque, there is a transition t_{2P} of T to P with $t_{2P} \prec_{\mathsf{hb}} t_2$ and t_{2P} justifying t_2. $t_{2P} \neq t_P$ by property of the deque: every transition to P can justify only one transition from P. t_2 is the first transition $(\prec_{\mathsf{to}})$-after t_P, so $t_{2P} \prec_{\mathsf{to}} t_P$. Absurd by H. So t is just after t_P by both $(\prec_{\mathsf{to}})$and $(\prec_{\mathsf{hb}})$, and reads from it: t respects H.

- If t is a *wakeup* transition.
  t is a Read-Modify-Write (RMW) operation, from a positive value to 0. So it must read from a store of a positive value to the same memory location. By examining the code, it is clear that all such stores are suspend transitions of the same task. So there is a suspend transition t_S of T with $t_S \xrightarrow{\mathsf{rf}} t$. As t_S is a release store and t is acquire, $t_S \prec_{\mathsf{hb}} t$. Since $\prec_{\mathsf{to}} \cup \prec_{\mathsf{hb}}$ is acyclic, $t_S \prec_{\mathsf{to}} t$ By the AtRMW axiom t_S is immediately before t in *mo* order.

Let's prove by contradiction that there is no t_2 with $t_S \prec_{\mathsf{to}} t_2 \prec_{\mathsf{to}} t$. We pick the earliest such t_2 (by $\prec_{\mathsf{to}}$). By H t_2 must be a wakeup transition since it is just after t_S, and it reads from t_S By the AtRMW axiom, t_2 is immediately after t_S in *mo* order. But that is t: Contradiction

- If t is a *repost* transition.
  By looking at the code it is clear that there is a wakeup transition t_W with $t_W \prec_{\mathsf{sb}} t$ (and so $t_W \prec_{\mathsf{to}} t$). Let's prove by contradiction that there is no t_2 with $t_W \prec_{\mathsf{to}} t_2 \prec_{\mathsf{to}} t$. By H, t_2 is a repost transition of T and is $(\prec_{\mathsf{sb}})$-after t_W. It is clear from the code that $t_W \prec_{\mathsf{sb}} t_2 \prec_{\mathsf{sb}} t$ is impossible. We have avoided having transition points in complex expressions, so all those by a single thread are comparable by $(\prec_{\mathsf{sb}})$. So $t \prec_{\mathsf{sb}} t_2$. So $t \prec_{\mathsf{hb}} t_2 \prec_{\mathsf{to}} t$. Which is in contradiction with the construction of $\prec_{\mathsf{to}}$.

□

Corollary 1. *All transitions of a task are ordered by* $(\prec_{\mathsf{hb}})$.

Theorem 14. *No data races can occur when running LibKPN, under the following hypotheses:*

- *a single task ever push to each queue;*
- *a single task ever pops from each queue;*
- *every task that is spawned is posted exactly once.*

Proof. We look at all the non atomic accesses:

- Accesses to the local variables of each task T
  Let a and b be two such accesses. By looking at the code of the scheduler loop, it is clear that there are two transitions t_{a1}, t_{a2} of T with $t_{a1} \prec_{\mathsf{sb}} a \prec_{\mathsf{sb}} t_{a2}$, and two transitions t_{b1}, t_{b2} of T with $t_{b1} \prec_{\mathsf{sb}} b \prec_{\mathsf{sb}} t_{b2}$, and t_{a2} is immediately after t_{a1} in $(\prec_{\mathsf{to}})$, and t_{b2} is immediately after t_{b1} in $(\prec_{\mathsf{to}})$. Three possibilities:
  - $t_{a2} \prec_{\mathsf{to}} t_{b1}$ Then $a \prec_{\mathsf{hb}} b$
  - $t_{b2} \prec_{\mathsf{to}} t_{a1}$ Then $b \prec_{\mathsf{hb}} a$
  - $t_{a1} = t_{b1}$ Then $t_{a2} = t_{b2}$ and a and b are in the same thread and cannot be part of a data race.
- Accesses to the queues' content
  By the same argument, accesses from the same side of the queue do not race with each other. Accesses from the producer and consumer end do not race by a property of the library we are using for the queues.
- Accesses to the stack pointer
  Because we take care to yield to the scheduler before doing the suspend or finish transitions, the same argument as for accesses to local variables hold.

□

6.3.3 Guarantee of progress

In order to prove that our algorithms neither deadlock nor livelock on valid KPNs we must define both progress and what it means for a KPN to be valid. We define progress as running in user code, and consider a KPN to be valid if there is a SC scheduling of it that does not get stuck.

Definition 22. *A task is said to be making progress if it is executing outside of the code of LibKPN.*

Definition 23. *A single-thread scheduling of a KPN is obtained by repeatedly picking one task that can make progress, and executing it for one task activation.*

Definition 24. *A KPN is said to be valid if there is a single-thread scheduling of it that can always find tasks ready to make progress, or that eventually has all tasks done.*

Lemma 4. *If a scheduler loop exits, every task whose spawn was* $(\prec_{\mathsf{hb}})$*-before the last read of* `num_spawned` *has a finish transition that was* $(\prec_{\mathsf{hb}})$*-before the last read of* `num_collected`*.*

Proof. If the scheduler loop exited, it means that the last loads of `num_spawned` and `num_collected` returned the same value n. For every task, there are three different possibilities when the scheduler loop exits:

- It is counted by neither `num_spawned` nor `num_collected` (for example because it has not been spawned yet).
- It has been finished, and its increment of `num_collected` was watched in the scheduler loop. Then it has necessarily been spawned, and by Corollary 1, this spawn transition happened-before the finish transition. As the load of `num_collected` is acquire, and its increment is release, the spawn transition also happened-before the load of `num_spawned`. So this task is counted in both loads.
- Its spawn transition was observed in the scheduler loop, but not its finish transition.

Since tasks in the third category increase the value read by the load of `num_spawned` but not the value read by the load of `num_collected`, and tasks in the other two categories do not change the difference between these two values, and these values had to be equal for the scheduler loop to exit, it is impossible for any task to be in the third category. □

Lemma 5. *If the scheduler loop exits, then every task whose spawn transition happened before the scheduler was started, and every task started by those tasks recursively have finished* $(\prec_{\mathsf{hb}})$*-before the exit of the scheduler loop.*

Proof. We say that a task has a *spawn depth* of 0 if it is spawned $(\prec_{\mathsf{hb}})$-before the start of the scheduler loop, and otherwise of $n+1$ where n is the spawn depth of the task that spawned it. We proceed by recurrence on the spawn depth d of each task T that verifies the hypothesis.
If $d = 0$, the spawn transition of T was obviously visible to the read of `num_spawned` in the scheduler loop. So by the previous lemma, it has finished $(\prec_{\mathsf{hb}})$-before the exit of the scheduler loop.
If $d = n + 1$, by recurrence hypothesis, T was spawned by a task T_0 that finished after the spawn, and before the load of `num_collected`. By Corollary 1, this implies that the spawn of T was $(\prec_{\mathsf{hb}})$-before the load of `num_collected`, and we conclude by the previous lemma. □

Lemma 6. *If the scheduler is stuck on every OS thread without waking up a task, then every task is in states U, S or F.*

Proof. We assume everywhere that if a task is spawned, it is immediately posted, so no task can remain stuck in state N.
All transitions that bring a task in state E or W are ($\prec_{\mathsf{sb}}$)-followed by transitions that take the same task out of this state. So U, S or F are the only remaining options. □

6.3.3.1 Proof of progress in algorithm F

Lemma 7. *When running algorithm F on a valid KPN, there are no spurious wake-ups. In other words, when a task T is woken up, it makes progress before suspending again.*

Proof. In algorithm F, the call to `tryawaken` is conditional on `is_not_full_or_not_empty` returning true on the same queue and with the same direction parameter. So at some point ($\prec_{\mathsf{sb}}$)-before the wakeup transition, there was data in the queue `q`. (assuming without loss of generality that T had suspended in a pop call). The queues are single-producer single-consumer, so only T could have removed elements from `q`. This can only happen while T is in the E state. It is impossible for the pop call to fail because of another pop call ($\prec_{\mathsf{hb}}$)-after it, so we only need to consider those before it. They are ($\prec_{\mathsf{hb}}$)-before the suspend transition (by Corollary 1), so they are ($\prec_{\mathsf{hb}}$)-before the call to `is_not_full_or_not_empty` and were fully visible to it. So there is still enough data in the queue for pop to succeed. □

Theorem 15. *When running algorithm F on a valid KPN, it is impossible for it to run forever without making progress, assuming a minimally fair scheduling (i.e. assuming that no thread is ever prevented from taking a step for an infinite amount of time).*

Proof. By contradiction: The only unbounded loop in LibKPN is the scheduler loop. If it finds tasks to take or steal, they will be able to make progress by the previous lemma. So the scheduler loop must never find a task to take or steal. By Lemma 6, every task is in one of states U, S or F. If every task is in states U or F, then `num_spawned` is equal to `num_collected` and the scheduler would have eventually exited. So there must be at least one task T_0 that is stuck in state S. This task is suspended on another task T_1. T_1 is either finished, or suspended on a task T_2, and so on. So either there is a task T_n that is suspended on T_{n+1} that is finished, or there is a cycle of tasks that are suspended on each other (as there is a finite number of tasks that can be spawned in a finite amount of time).

- In the first case, T_{n+1} has pushed enough data for T_n to read, or it would not be a valid KPN. Thanks to the presence of a sequentially consistent fence in both `push` and `suspend_done`, either T_{n+1} loaded a positive value for `waitvar` in `tryawaken` and will wake-up T_n, or T_n saw that enough data was available in the call to `is_not_full_or_not_empty` and cancelled its own suspension. In both cases, T_n cannot remain stuck in the suspended state.

- In the second case, consider the amount of data pushed and pop on each channel in the loop. If all suspensions were justified (i.e. there is no space left for any push, nor any data left for any pop), then we would reach the same deadlock with a single-thread scheduling, and the KPN is not valid. So there is at least one suspension that is unjustified: that is a T_n that is suspended on a pop on a channel from T_{n+1}

that has data (we only consider pop, as push is similar). Like above, this would require T_n not to see the last push from T_{n+1} on this channel, and T_{n+1} not to see the suspension of T_n in its push; and this is impossible thanks to the fences in `push` (for T_{n+1}) and `suspend` (for T_n).

□

Remark: The requirement of a fair scheduling of the scheduler threads shows that despite being built from lock-free data structures, LibKPN as a whole is not lock-free.

6.3.3.2 Proof of progress in algorithm S

Lemma 8. *In a minimally fair implementation, it is impossible for no task to go through any transition for an infinite time without the scheduler exiting or a task executing.*

Proof. By contradiction. By a proof analogous to that of 15, we must have a cycle of tasks suspended on each other. $(\prec_{\mathsf{sc}})$is a total order on the fences in these calls to `suspend`. So there must be two tasks A and B, with the fence in the suspension of A $(\prec_{\mathsf{sc}})$-before the one in the suspension of B and B suspended on A. But in that case, the scan in the suspension of B would have seen the suspension of A, and would have woken up A. □

Lemma 9. *It is impossible for a task T_0 to have an infinite number of wakeup transitions without some task making progress.*

Proof. These wake-up transition cannot come from `push`/`pop`, or they would lead to progress. So they must all come from the scan in the suspension of another task T_1. Since each suspension is preceded by a wake-up, that task must itself have been woken up each time by a suspension from a task T_2, and so on. We build this way a loop of tasks repeatedly suspending, waking up the next task in the cycle, and being woken up again. Because the synchronisation on `q->wait[d]` is release/acquire, it creates synchronises-with edges, and at some point all tasks in the cycle can see all the progress ever done by all other tasks. If they still cannot make progress, this cycle would have caused a deadlock in an SC implementation, and the KPN is invalid. □

Theorem 16. *When running algorithm S on a valid KPN, it is impossible for it to run forever without making progress, assuming a minimally fair scheduling (i.e. assuming that no thread is ever prevented from taking a step for an infinite amount of time).*

Proof. Trivial by the previous two lemma. □

6.3.3.3 Proof of progress in algorithm H

We consider only the last version of the algorithm, with support of select and no spurious wake-ups (Section 6.2.4).

Lemma 10. *When running algorithm H on a valid KPN, there are no spurious wake-ups. In other words, when a task T is woken up, it makes progress before suspending again.*

Proof. Here are the ways a task can be woken up in algorithm H:

- By the calls to `tryawaken` in `push` and `pop`.
- By the first call to `tryawaken` in `suspend` or `suspend_select`, that cancels a suspension.

- By the call to `tryawaken` in the scanning part of `suspend`, `suspend_select` or `suspend_done`.

In all three cases, the call to `tryawaken` is guarded by a check of `is_not_full_or_not_empty` with the same queue and direction. At this point the proof is almost the same as that of Lemma 7 (that proves the same property for algo F).

The only new case is that of a suspension caused by select. In that case, the suspended task may no longer be waiting on the same queue. This is not a problem because in that case the compare-and-exchange in `tryawaken` will fail (see Section 6.2.4). □

Theorem 17. *When running algorithm H on a valid KPN, it is impossible for it to run forever without making progress, assuming a minimally fair scheduling (i.e. assuming that no thread is ever prevented from taking a step for an infinite amount of time).*

Proof. This proof proceeds similarly to the proof for algorithm F (Theorem 15), relying on the same property of the absence of spurious wake-ups. We show in the same way that all tasks must be in one of the U, F or S states. If none of the suspended tasks are suffering from a wrong suspension (i.e. trying to pop/select from a queue that has data in it), then we could reach the same stalemate in a sequential execution: the KPN is not valid So there are two tasks A and B and a queue q from A to B, with B suspended on a pop/select from q and enough data pushed (by A) into q to satisfy it. A cannot be in state U (since it pushed data), so it is either in F or S. Consider its last call to `suspend`, `suspend_select` or `suspend_done`. It has a sequentially consistent fence, call it f. If f_A is $(\prec_{\mathsf{sc}})$-before the fence f_B in the last call to `suspend` or `suspend_select` of B, then B would have seen the push by A and cancelled its suspension. Reciprocally, if f_B is $(\prec_{\mathsf{sc}})$-before f_A, then A would have seen that B is wrongly suspended in its scan and would have woken it up. □

6.4 Remarks on C11 programming

We have seen how to start from a sequentially consistent algorithm and progressively relax it for performance. As we did so, we had to change not just the attributes on the atomic accesses, but the algorithmic details, and the invariant respected by the algorithm to avoid fences in the critical paths.

Several patterns were recognisable:

- A release store synchronising with an acquire load to create the happens-before relation. This corresponds to the MP pattern (Figure 2.1). This is used to make the activations of each task ordered, through the `release` in suspend, and the acquire in `tryawaken`.

- Two threads each write to a different variable, and we want at least one to notice the store by the other thread. This is achieved by adding a sequentially consistent fence between the store and the load in each thread. It corresponds to the SB pattern (Figure 1.1). In the algorithm F, it appears as shown in Figure 6.5. In algorithms S and H, it appears between calls of `suspend` by adjacent tasks.

- Finally, when two threads might notice that a task must be woken up, and we want only one to do it (to avoid duplicating the task), we use either an exchange or compare-and-exchange operation. This is used in `tryawaken` in all three algorithms.

Interestingly, while the algorithms were designed mostly through these basic building blocks, their proof of correctness is built by induction, and not modular. It has been previously noted that the C11 model does not lend itself to modular proofs [BDG13, Lê16], and it remains an open question how to do it.

6.5 Applications and experimental evaluation

The work presented in this section is not my own, it was mostly done by Nhat Minh Lê and Adrien Guatto. I include this section as evidence that the design of LibKPN is efficient in practice.

We present three sets of benchmarks:

- A few micro-benchmarks to compare the algorithms F and S. Sadly, there has not been time to implement (and benchmark) the algorithm H, it is still work in progress.
- Two dataflow benchmarks to compare ourselves with a specialized language such as StreamIt that statically schedules a subclass of KPNs.
- Two linear algebra benchmarks, to evaluate our overhead compared to OpenMP in the worst case (less than 5%), and to show that KPNs are a model more widely applicable than usually believed.

These benchmarks were run on the following machines:

i7. A 4-core desktop with an Intel Core i7-2600 at 3.4 GHz, Linux version 3.15.

Xeon. A 12-core dual-socket workstation with two Intel Xeon E5-2300 at 2.6 GHz, Linux version 3.13.

Opteron. A 24-core blade with two AMD Opteron 6164 HE at 1.7 GHz, Linux version 3.2.

POWER. 16 cores out of 32 on an IBM POWER 730 with four POWER7 processors running at 3.55 GHz, and Linux version 3.8. The machine is shared, with only half of the processors available for benchmarking purposes.

A15. A 2-core Arndale board with a Samsung Exynos 5250 SoC built on an ARM Cortex 15 MPCore at 1.7 GHz, Linux version 3.13, throttled at 1 GHz for thermal issues.

The linear algebra kernels rely on the MKL to provide fast BLAS routines and thus were evaluated on x86 only.

We used ICC version 14 with options `-O3 -ipo -xHost` where possible, or GCC 4.9.1 with options `-Ofast -march=native` on Opteron and ARM, and `-Ofast -mcpu=power7` on POWER. In particular, at the time of writing, SMPSs, Swan and OpenMP 4 require GCC.

6.5.1 Micro-benchmarks

We considered three different KPN topologies:

map reduce a classic fork-join pattern. On all platforms, algorithm S is faster than algorithm F. It is about twice as fast on the Opteron machine.

clique a fully connected graph of tasks. Once again, algorithm S is faster than algorithm F, but the gap is very small now. This was expected, as the suspend operation in algorithm S has a cost that depends on the degree of the task.

cycles a small number of long pipelines that loop back on themselves. On most runs, algorithm S is still faster, but we can observe here some pathological behaviours. More precisely, when there is a small number of very long pipelines, the runtime of algorithm S becomes extremely variable, occasionally up to orders of magnitudes higher than for algorithm F.

This can easily be explained: algorithm S depends on a cascade of spurious wake-ups to reach tasks that were missed in a push or pop and wake them up. In this specific case, bad timing can cause this cascade to occur to every (or most) pushes and pops, and it must loop all over the (arbitrarily long) pipeline. This rare pathological behaviour motivates the design of algorithm H that strives to keep the low overhead of algorithm S (shown useful on the other benchmarks) while avoiding spurious wake-ups.

We also compared our implementation with two others: a reference Go implementation (in Go 1.3) using goroutines and go channels, and an implementation using Linux native threads with FIFO queues from the high-performance Intel TBB library. The Go code was always slower than our algorithm F, often by a factor of about 5. Using native threads was even worse, with a slowdown of 89 times on mapreduce.

The following caveat should be noted: both Go and TBB implement multiple-producer multiple-consumer (MPMC) queues, which are intrinsically more costly than their SPSC counterparts.

6.5.2 Dataflow benchmarks

6.5.2.1 FMRadio

We first consider FMRadio, the prototypical StreamIt benchmark [TKA, TA], inspired by software radio applications. It is representative of signal processing workloads with pipelines of data-parallel filters.

Any comparison against StreamIt itself is complicated by the fact that their compiler outputs specialized C code depending on the characteristics of the target machine [GTA], most importantly, the number of worker threads requested. It statically partitions and schedules the program dataflow graph, which makes versions compiled for different number of processors difficult to compare. In this test, we evaluate two codes generated by StreamIt: a sequential version, and a 4-core version. Accordingly, we also set up our runtime with only four threads. The StreamIt programs have widely differing performance curves depending on the platform, due to the static work estimation and partitioning being—in all likelihood—mostly tuned for x86. They are mostly presented here for reference.

This experiment shows in Figure 6.10 that there are benchmarks larger than the synthetic ones we showed earlier where the algorithm S offers a significant speedup (up to 2) over the algorithm F.

6.5.2.2 LDPC

Our second dataflow application is a Low-Density Parity Check (LDPC) decoder, which implements the cheapest form of iterative belief propagation. Such error-correction codes are ubiquitous in high-rate wired and radio communications. Interestingly, the algorithm is not often thought of as belonging to the family of dataflow algorithms. Rather, most implementations are imperative and iterative. However, the computation can be elegantly described in terms of bouncing messages between nodes of a bipartite graph. At each iteration, every vertex reads from all its neighbors, computes a confidence index, and propagates back this index to its neighbors. The computation stops when a specific criterion called the *syndrome* is satisfied.

This naturally leads to an implementation based on tasks aggregating vertices across iterations, and channels to represent the communications between them. Depending on topology of the coding graph, the algorithm will exhibit more or less parallelism. Sparser graphs and less connected components lead to more potential parallelism.

We compare our dataflow implementation to a reference code provided by Thales C&S to evaluate future hardware platforms for mobile and *ad hoc* radio communications. This reference implementation is written in OpenMP following the iterative algorithm, using *parallel for* loops and synchronization barriers.

The results in Figure 6.11, against a fixed matrix provided by Thales C&S, show no clear winner between Algorithms F and S. This is because, although our version of LDPC is written in a stream-oriented style, with long-running macro-tasks and channels, pipelines are typically very short on this data set, spanning at best one or two iterations.

Compared to the OpenMP implementation by THALES, our dataflow LDPC performs better on every platform, though the difference is less on A15 and i7. The three other machines offer more processors, hence more opportunities for schedule gaps to form, while a thread is waiting for an OpenMP barrier, something the dataflow version does not suffer from.

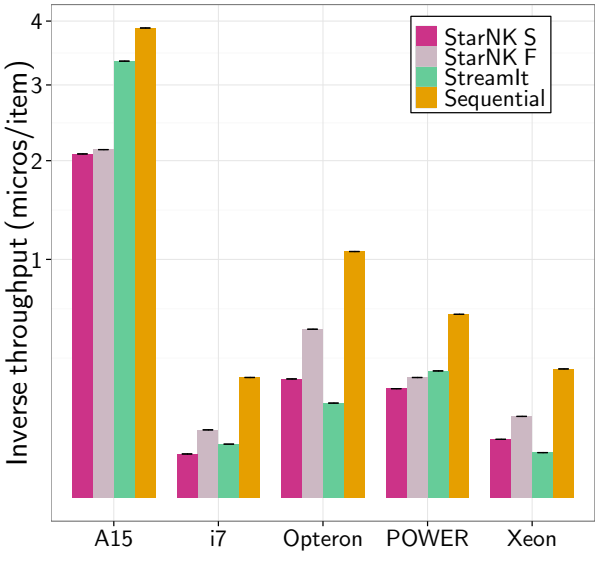


Figure 6.10: FMRadio ("StarNK" is the previous name of LibKPN)

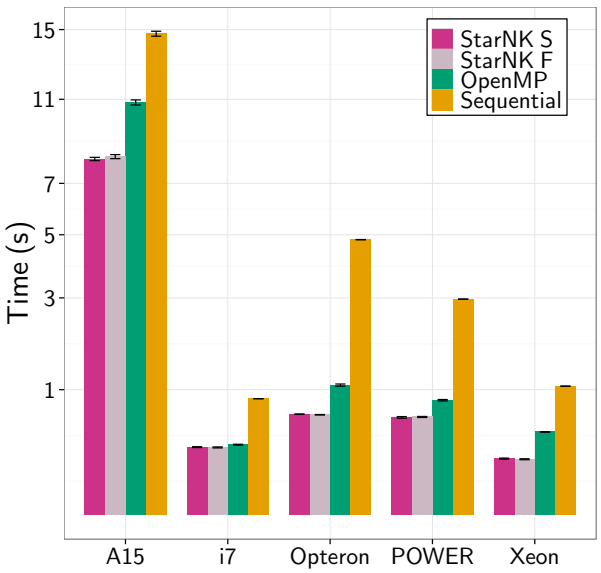


Figure 6.11: LDPC

6.5.3 Linear algebra benchmarks

6.5.3.1 Cholesky factorization

The Cholesky matrix factorization is a well-known problem, with practical high-performance parallel implementations readily available in LAPACK libraries such as PLASMA [BLKD09] and Intel MKL. Efficient parallel versions of Cholesky are based on the tiled algorithm. The input matrix is divided into tiles, and matrix operations are applied on them. Each output tile is the result of a series of operations on both input and previously computed output tiles. If each of these operations is a task, then the task graph is clear: a task depends on the tasks that complete the computation of its required operands (if they are output tiles).

In a first time, we reused an OpenMP 4 implementation through an OpenMP 4 emulation layer (labeled *KOMP*) on top of our runtime. Interestingly, once the performance curves plateau (i.e. for large matrices), KOMP is within 5% of the performance of the OpenMP 4 runtime. For simplicity, we have opted for a naive dynamic interpretation of OpenMP directives in our emulation layer, so part of the overhead can be attributed to the cost of mapping and juggling sequentially with dependency objects within the parent task. More importantly, the difference between KOMP and OpenMP is within the variations of Swan and SMPSs, which perform alternatively better and worse than OpenMP on the two platforms. Therefore we conclude that allocating and managing individual dependency objects do not incur any incomparably high cost.

In a second time we wrote an optimised version of the same algorithm, taking advantage of our richer model to aggregate tasks as much as possible. We won't cover the specific details of this implementation here as it is not relevant to this thesis. Figures 6.12 and 6.13 show the fruit of our efforts. On our test platforms, our enhanced version, labeled *KPN*, consistently outperforms the competition, including the state-of-the-art Intel MKL and PLASMA implementations. All three test codes have been compiled using the Intel C Compiler with appropriate options, and are backed by the MKL BLAS and LAPACK. On the bigger 12-core Xeon machine, the difference is more visible. Our enhanced algorithm peaks earlier, exhibiting a throughput 19% higher at sizes 4000 and 5000, which progressively tapers off; still, our FLOPS count remains 5% higher than the closest alternative once all implementations plateau.

The MKL results deserve their own small paragraph. Contrary to other implementations of Cholesky, the MKL automatically selects its own tile size. Furthermore, it is unknown what algorithms and schedulers are actually used by the library. As such, we can only speculate as to the reason of the sharp drop in performance around 7000. The most likely explanation is that smaller sizes use dedicated settings, ad hoc implementations, partially static schedules, or a combination thereof, which are abandoned at higher matrix sizes in favor of more generic but maybe less well-tuned algorithms.

6.5.3.2 LU factorization

As a complementary result, Figures 6.14 and 6.15 present FLOPS measurement for five implementations of the LU factorization: the SMPSs version and its OpenMP 4 port, the MKL and PLASMA parallel LAPACK routines, and our own version.

Our algorithm is based on the ScaLAPACK task graphs [CDO+96]. In this respect, it is similar to the SMPSs code. However, the latter implements a simpler variant where synchronization barriers are used to divide the computation into big steps, with fine-

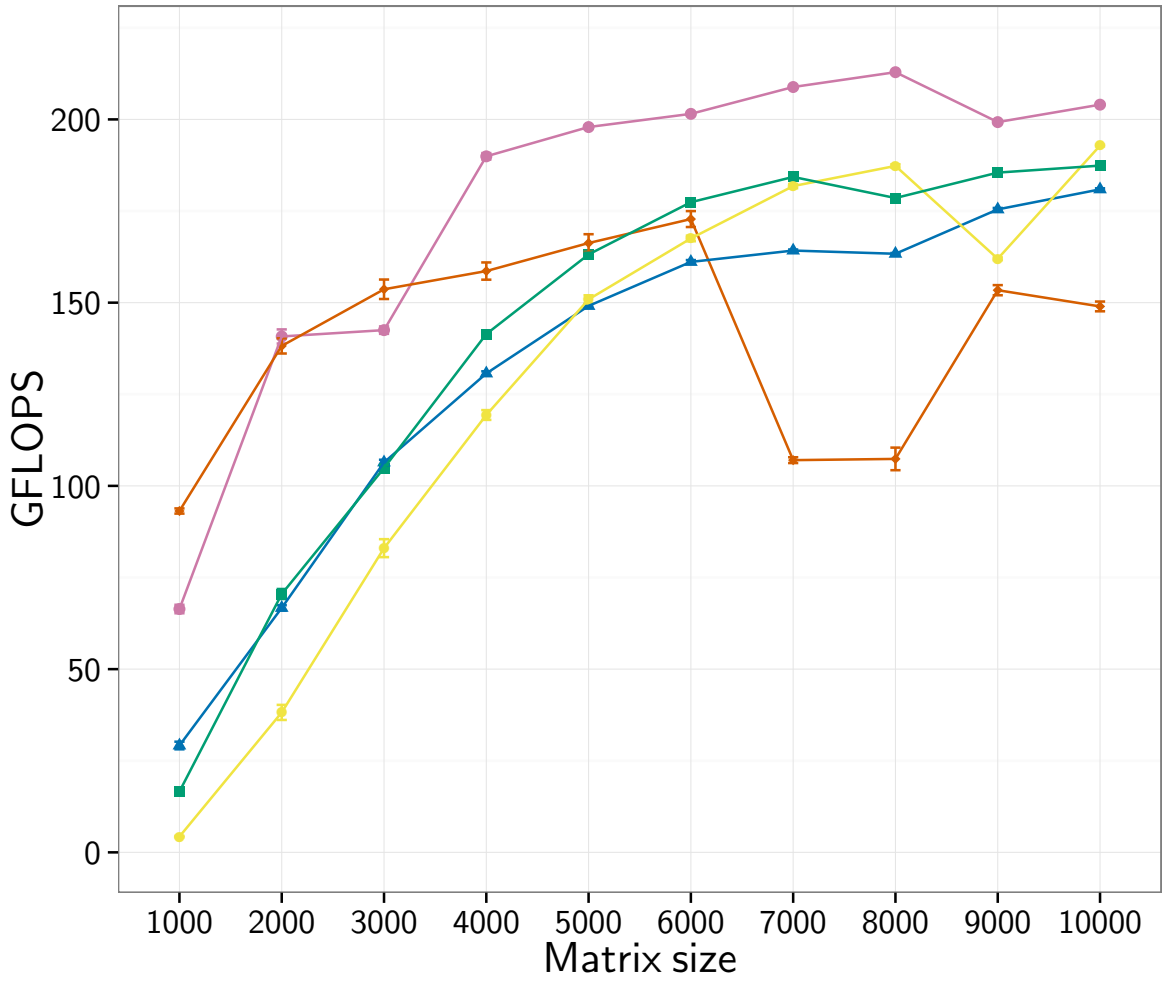


Figure 6.12: Cholesky factorization on Xeon (see Figure 6.14 for legend)

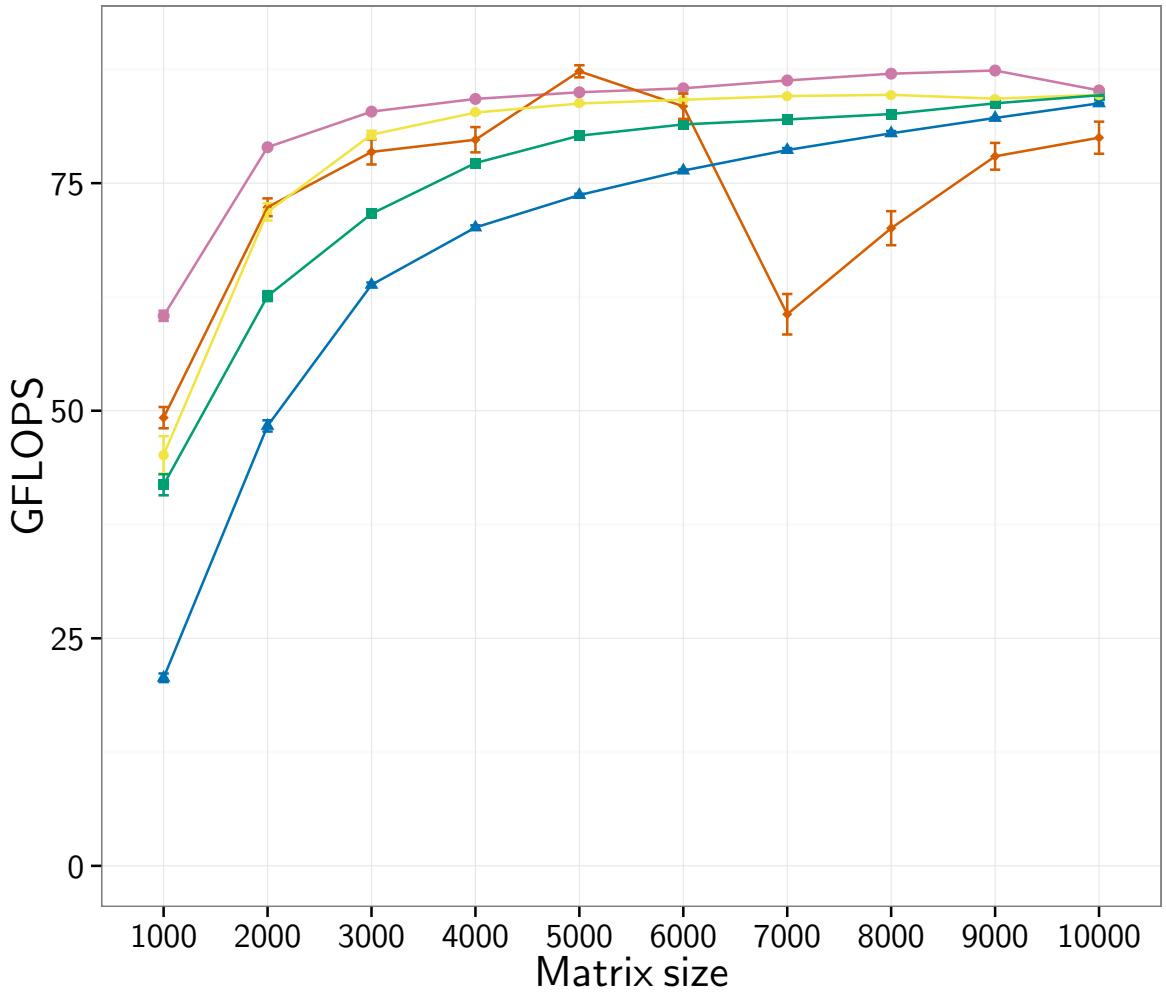


Figure 6.13: Cholesky factorization on i7

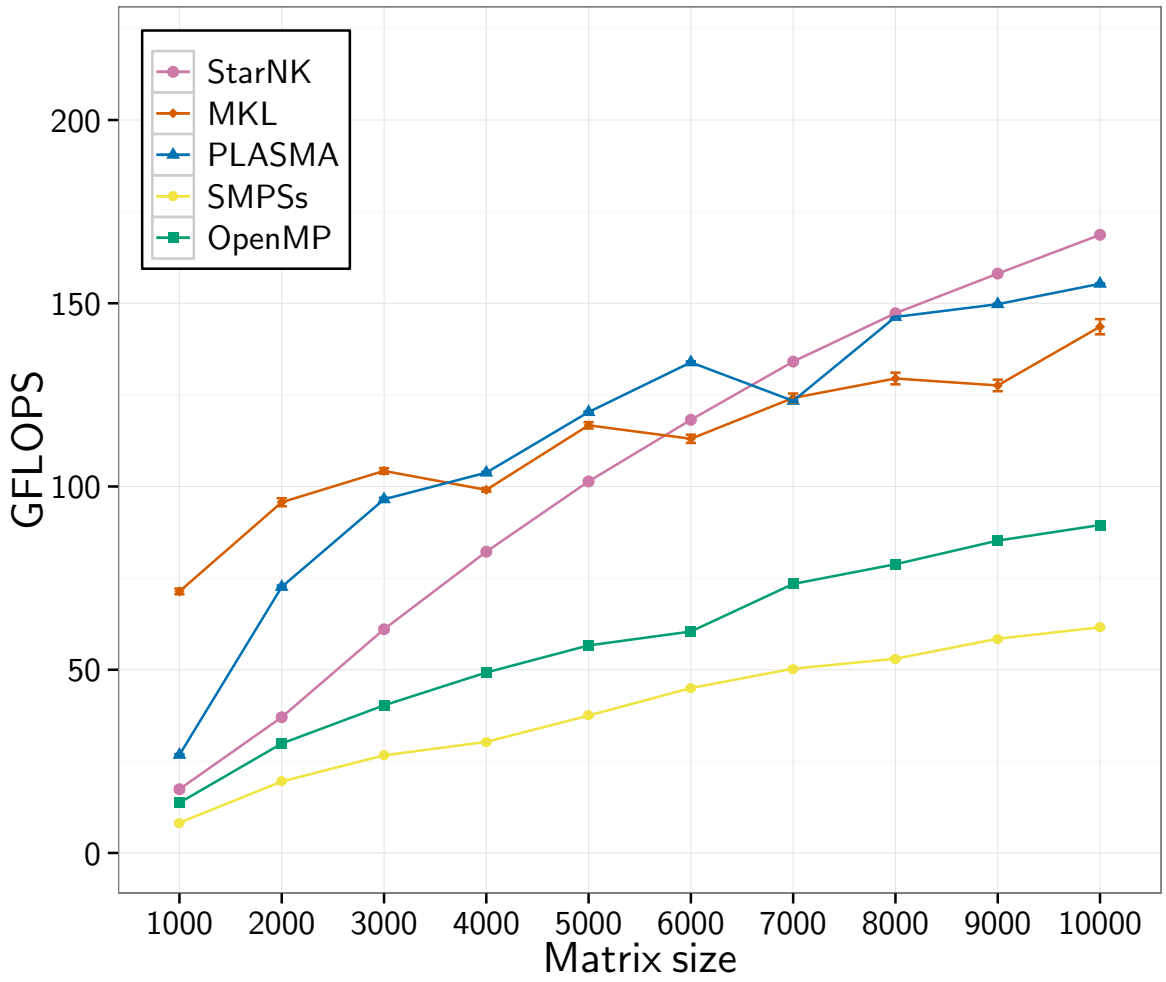


Figure 6.14: LU factorization on Xeon

grained dependencies in each step. This prevents these versions from scaling on the higher number of cores of the Xeon machine.

Compared to the MKL and PLASMA, our implementation appears to be competitive on moderate and large matrices, but suffers on smaller sizes. While we applied similar optimization techniques to LU as we did to Cholesky, our implementation is missing one improvement that we believe is mostly responsible for these relatively poor results: in the LU factorization, the *getf2* step that is fairly expensive can execute on large portions of the matrix sequentially. We believe it would benefit from parallelization at a finer grain, rather than being treated as a whole tile.

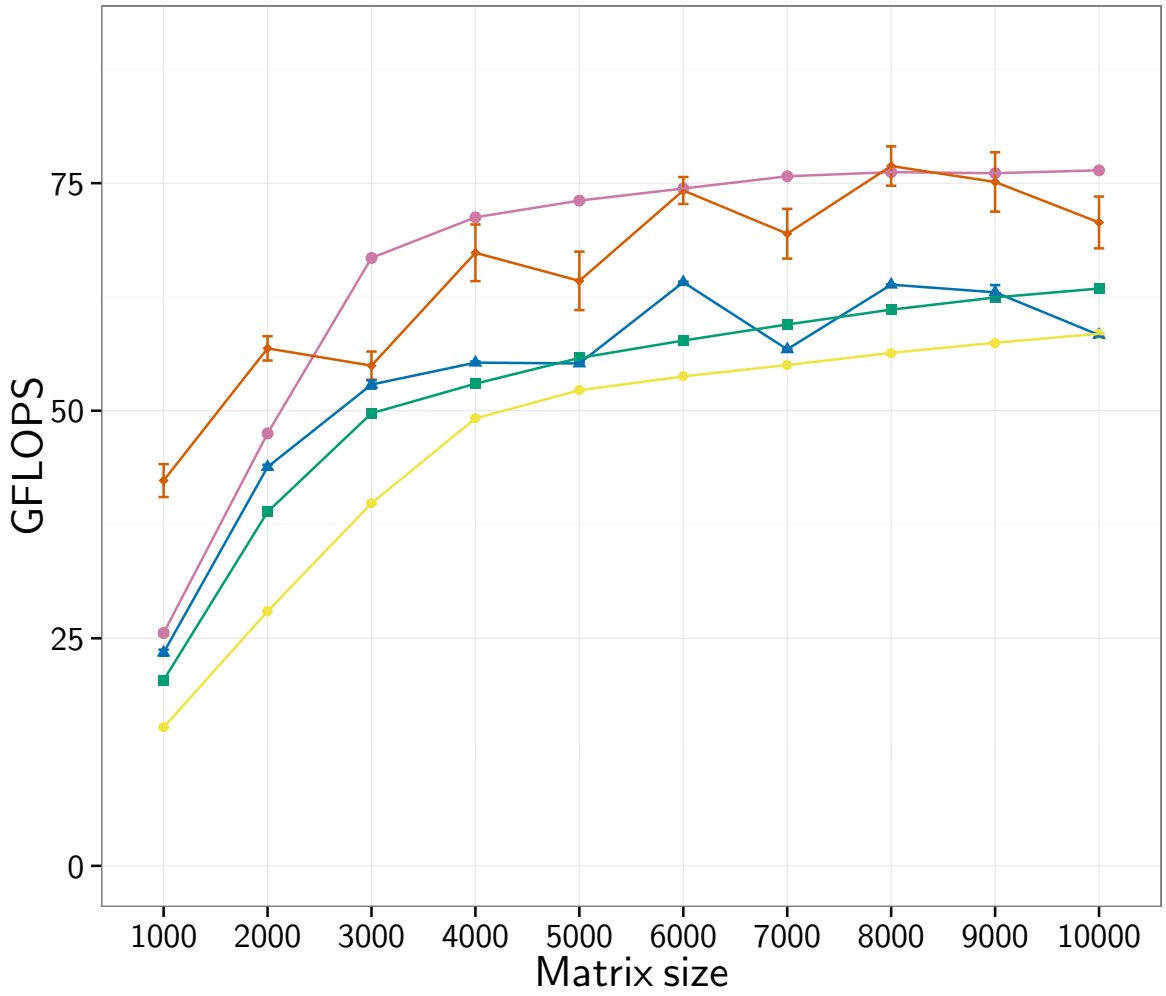


Figure 6.15: LU factorization on i7

Chapter 7

A fence elimination algorithm for compiler backends

Currently, compilers translate atomics in a completely local fashion, following the mappings published on [C11]. The recommended mapping for an acquire load is to add a branch and an isb fence (Instruction Synchronisation Barrier) after the load; but Clang prefers the equally correct use of a `dmb ish` (Data Memory Barrier over the Inner SHareable domain) fence after the load as it is faster on at least some processors. So the following code:

```
r = x.load(acquire);
y.store(release, 42);
```

gets translated to the following pseudo-assembly:

```
r = x;
dmb ish; // fence acquire
dmb ish; // fence release
y = 42;
```

We gave a precise account of the ARM memory model in Section 2.2.2, it is easy to show that whenever there are two consecutive `dmb ish` instructions, the second acts as a no-op. As such it can safely be optimised away without affecting the semantics of the program in an arbitrary concurrent context. This simple peephole optimisation has already been implemented in the LLVM ARM backend [Elh14]. However, more generally, the compiler backend can rely on *thread-local data-flow* information to identify occasions to optimise the generated code while preserving the expected semantics. Consider for instance the snippet of C11 code below:

```
int i = x.load(memory_order_seq_cst);
if (foo())
  z = 42;
y.store(1, memory_order_release);
```

and the ARMv7 pseudo-assembly generated by the mappings for atomic accesses:

```
bool a = foo();
int i = x;
dmb ish;
if (a)
  z = 42;
```

```
dmb ish;
y = 1;
```

In the above code neither of the two `dmb ish` barriers can be eliminated. Intuitively, the former fence ensures that the load of `x` is not reordered with the store of `z`, and the latter fence ensures that the write to `z` is visible by all processors before the write to `y` is performed. However this fence placement is *not optimal*: when `a` is false two barriers in a row would be executed while one would suffice (to prevent the reordering of the load of `x` with the write of `y`), and the pseudo-code could be optimised as:

```
bool a = foo();
int i = x;
dmb ish;
if (a) {
  z = 42;
  dmb ish;
}
y = 1;
```

Now, the optimised pseudo-assembly cannot be obtained by applying a source-to-source transformation to the original C11 program and then applying the reference mappings. This suggests that a class of fence optimisations must be implemented in an architecture-specific backend of a compiler, exploiting the precise semantics of the target architecture. However, apart from some peephole optimisations like the already cited one [Elh14], at the time of writing mainstream compilers are very conservative when it comes at optimising atomic accesses or reordering atomic and non atomic accesses.

Contributions In this chapter we show how to instantiate Partial Redundancy Elimination (PRE) to implement an efficient and provably correct fence optimisation algorithm for the x86, ARM and IBM Power architectures, improving and generalising the algorithm proposed for x86 in [VZN11]. Although these architectures implement widely different memory models, we identified a key property required for the correctness of our algorithm and we prove formally that our algorithm is correct with respect the memory model of the three architectures. We have implemented our optimisation algorithm in the x86, ARM, and Power backends of the LLVM Compiler Infrastructure [LA04]. We evaluate it on some benchmarks, including a signal processing application from the StreamIt suite, on which we measure observe up-to 10% speedup on the Power architecture.

7.1 A PRE-inspired fence elimination algorithm

The design of our fence elimination algorithm is guided by the representation of fences in the “Herding Cats" framework [AMT14] for weak memory models. This framework, detailed in Section 2.2.2, represents all the allowed behaviours of a program in terms of the sets of the memory events (atomic read or writes of shared memory locations) that each thread can perform, and additional relations between these events. Fence instructions do not generate events of their own, but are represented as a relation between events. This relation relates all pairs of memory accesses such that a fence instruction is executed in between by the sequential semantics of each thread.

This implies that any program transformation that moves/inserts/removes fence instructions while preserving the fence relation between events is correct. We can even go

a bit further. The model is monotonic with respect to the fence relation: adding pairs of events to the fence relation can only reduce the allowed concurrent behaviours of a program. In turn any program transformation modifying the placement of fence instructions will be correct provided that it does not remove pairs from the fence relation between events.

Consider now the snippet of C11/C++11 code below, where `x` and `y` are global, potentially shared, atomic variables and `i` is a local (not shared) variable:

```
int i = x.load(seq_cst);
for (; i > 0; --i) {
  y.store(i, seq_cst);
}
return;
```

and the ARMv7 pseudo-code[1] generated by desugaring the loop and applying the mappings for the atomic accesses:

```
int i = x;
dmb ish;
loop:
  if (i > 0) {
    dmb ish;
    y = i;
    dmb ish;
    --i;
    goto loop;
} else {
    return;
}
```

A sequential execution of this code with `x = 2` executes the following instructions:

```
i = 2;  dmb ish;  dmb ish;  y = 2;  dmb ish;  i = 1;
  dmb ish;  y = 1;  dmb ish;  i = 0;  return
```

resulting in two events related by the fence relation (omitting the fence arrows from the entry point and to the return point):

If one fence is removed and the program transformed as in:

```
int i = x;
loop:
  dmb ish;
  if (i > 0) {
    y = i;
    --i;
```

[1]Our optimisation runs on the LLVM IR at the frontier between the middle end and the architecture specific backend. This IR extends the LLVM IR with intrinsics to represent architecture specific memory fences; the pseudo-code we rely-on carries exactly the same information as the IR over which our implementation works.

```
    goto loop;
  } else {
    return;
}
```

then a sequential execution of this code with `x = 2` executes the following instructions:

```
i = 2;  dmb ish;  y = 2;  i = 1;
  dmb ish;  y = 1;  i = 0; dmb ish; return
```

and both the events generated and the fence relation remain unchanged, and we are guaranteed that the two programs will have the same behaviours in any concurrent context. However the latter will execute fewer potentially expensive `dmb ish` instructions in any run.

7.1.1 Leveraging PRE

At a closer look, our problem is reminiscent of the *Partial Redundancy Elimination* (PRE) optimisation. Given a computation that happens at least twice in the source file (say `x+y`), PRE tries to insert, move or delete instances of the computation, while enforcing the property that there is still at least one such instance on each path between the definitions of its operands (say `x` and `y`) and its uses. Similarly, we want to insert, move or delete fences such that there is still at least one fence on each path between memory accesses that were before a fence and those that were after a fence in the original program.

How to implement PRE is a well-studied problem. Approaches can be classified into *conservative* (roughly speaking, the amount of computation the program does cannot be increased) and *speculative*. Speculative PRE (SPRE) is allowed to introduce some extra computation on rarely taken paths in the program to remove additional computations from the more common paths. We follow this latter approach because it can offer better results [HH97] in presence of accurate profiling information. Such *profile-guided optimisations* require compiling the program under consideration twice: first without optimisations, then a second time after having executed the program on a representative benchmark with profiling instrumentation on. Our algorithm can be adapted to avoid this complex process at a cost in performance, as discussed in Section 8.2.2.

We build on the elegant algorithm for SPRE presented in [SHK04] (later generalised to conservative PRE in [XK06]), based on solving a *min-cut* problem. Given a directed weighted graph (V, E) with two special vertices *Source* and *Sink*, the *min-cut* problem consists in finding a set of edges $C \subseteq V$ such that there is no path from *Source* to *Sink* through $E \backslash C$ and C has the smallest possible weight.

Following [SHK04], our algorithm first builds a graph from the SSA control-flow graph internally used by LLVM. In the built graph, nodes identify all program placements before and after each instructions and there is an edge from after instruction A to before instruction B with weight w if control went directly from A to B w times in the profiling information. Two special nodes are added, called *Source* and *Sink*. Edges with weight ∞ are added from the *Source* node and to the *Sink* node, following the strategy below.

For each fence instruction in the program,

- connect all the placements after all memory accesses that precede the fence to *Source*;
- connect all the placements before all memory accesses that follow the fence to *Sink*;

- delete the fence instruction.

Once all fences have been deleted, a min-cut of the resulting graph is computed: the min-cut identifies all the places where fences must be inserted to guarantee that there is still a fence across every computation path between memory accesses originally separated by a fence. Each edge in the min-cut identifies two placements in the program, the former after a memory access and the latter before another memory access: the algorithm inserts a fence instruction just before the latter memory access.

Algorithm 1 reports the details of our implementation in LLVM, the entry point is the function `TransformFunction`. Our optimisation runs after all the middle-end transformations; at this level, the IR is in SSA form but includes the architecture-specific intrinsics for fence instructions which have been inserted by the expansion of the C11/C++11 atomic memory accesses. It differs slightly from the description above inasmuch. Instead of creating two nodes for every instruction, it lazily constructs nodes (with the LLVM `GetNode` functions) at the beginning and end of basic blocks, and before and after memory accesses. Considering only these points is correct, because fences commute with instructions that do not affect memory. The `GetNode` functions refers to a hashtable that map locations in the code to nodes: if a node has already been generated for a location, a pointer to that same node is returned, it is not duplicated.

7.1.2 The algorithm on a running example

Let us illustrate our algorithm on the code snippet shown at the beginning of this section, assuming that some profiling data are available. Its control-flow graph (CFG) is shown in Figure 7.1a.

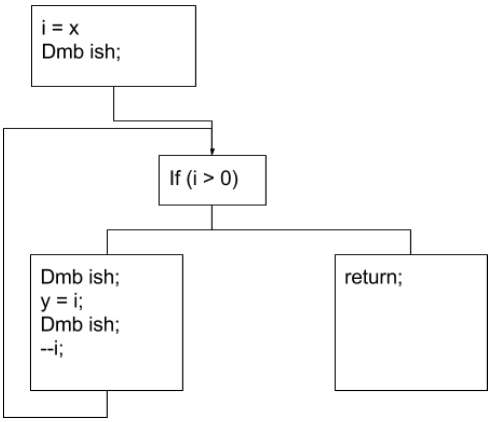
In the first phase, the algorithm considers each fence in order and builds an ad-hoc flow graph that for each fence connects all the memory accesses before the fence to all the memory accesses after the fence. The terms "before" and "after" are relative to the original control-flow of the function.

In the running example, after processing the first fence, we get the flow-graph in Figure 7.1b. The node in green (after `i = x`) is connected to the *Source* and the nodes in red (before `y = i` and `return`) are connected to the *Sink*. The percentages next to the edges are weights, set proportionally to the frequency of execution of these edges in the profiling data. After processing all the fences, we obtain the flow-graph in Figure 7.1c (where the node after `y = i` is connected to the Source).

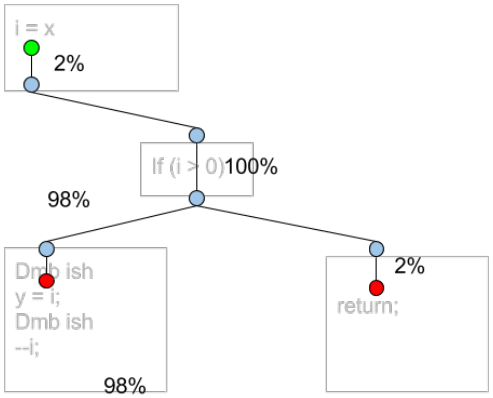
At this point, all fences are removed from the program and the min-cut is computed. Figure 7.1d shows one possible min-cut on this graph (through bold red edges).

In the second phase fences are inserted on each edge that is part of the cut. The resulting optimised CFG is shown in Figure 7.1e.

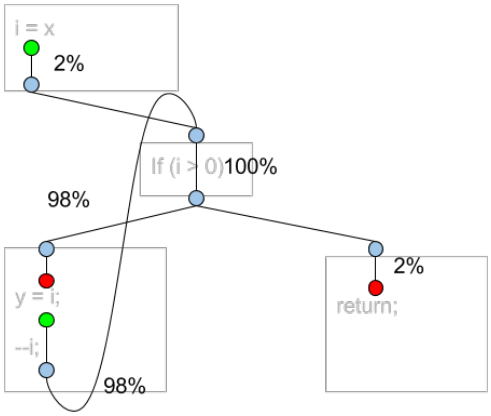
This algorithm generates a solution that is optimal in terms of the number of executions of fences at runtime; in some cases we might instead minimize the code size. As explained Section 8.2.2, we can modify this by modifying artificially the weights on the edges. In particular, adding a tiny constant C to each edge breaks ties among possible min-cuts according to the number of fences in the generated code. This approach is shown in Figure 7.1f, with the resulting CFG in Figure 7.1g; observe that here the fence in the first block has been removed.



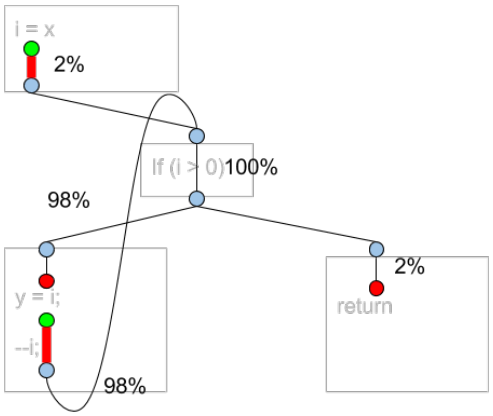
(a) Original CFG



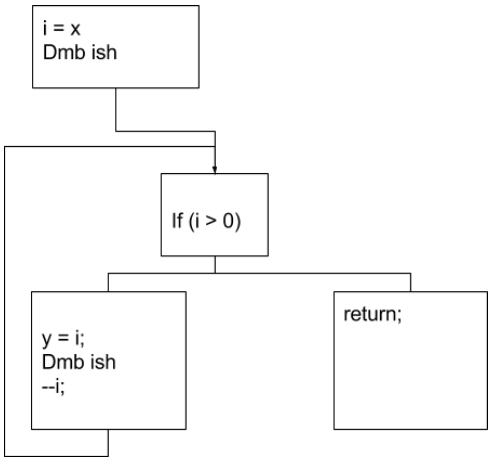
(b) After processing the first fence



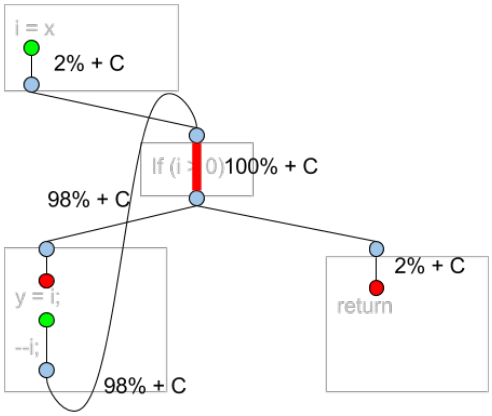
(c) After processing all fences



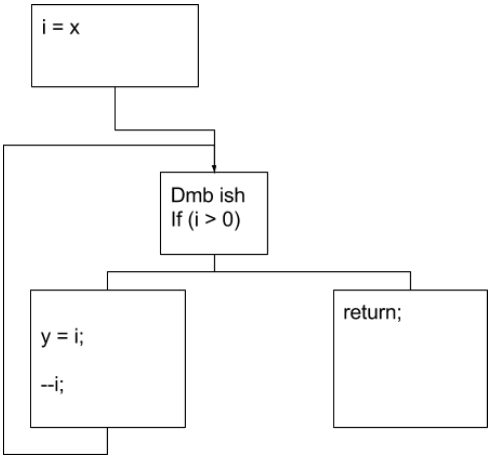
(d) Example of a min-cut



(e) Resulting CFG



(f) Min-cut to minimise code-size

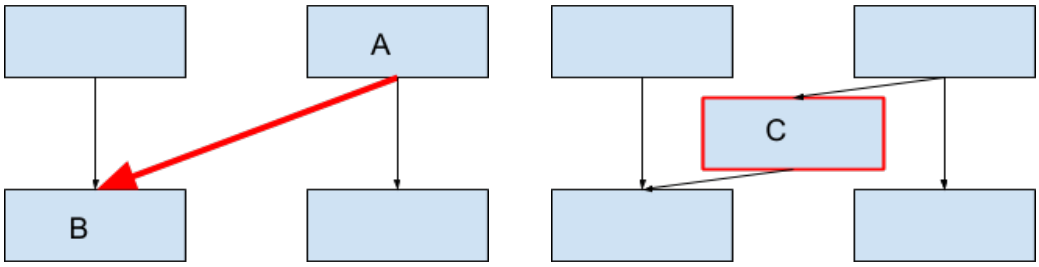


(g) Resulting CFG

Figure 7.1: Running example of the fence elimination algorithm

```
1  Function TransformFunction(fun)
2      for f a fence in fun do
3          nodeBeforeFence ←MakeGraphUpwards(f)
4          nodeAfterFence ←MakeGraphDownwards(f)
5          if nodeBeforeFence ≠ NULL && nodeAfterFence ≠ NULL then
6              MakeEdge (nodeBeforeFence, nodeAfterFence)
7          DeleteFence (f)
8      cuts ←ComputeMinCut ()
9      foreach location ∈ cuts do
10         InsertFenceAt(location)
11 Function MakeGraphUpwards(root)
12     basicBlock ←GetBasicBlock(root)
13     for inst an instruction before root in basicBlock, going upwards do
14         if inst a memory access then
15             node ←GetNodeAfter(inst)
16             ConnectSource(node)
17             return node
18     node ←GetNodeAtBeginning(basicBlock)
19     if basicBlock is first block in function then
20         ConnectSource(node)
21         return node
22     for basicBlock2 a predecessor of basicBlock do
23         node2 ←GetNodeAtEnd(basicBlock2)
24         inst2 ←GetLastInst(basicBlock2)
25         node3 ←MakeGraphUpwards(inst2)
26         if node3 ≠ NULL then
27             MakeEdge(node3, node2)
28             MakeEdge(node2, node)
29 Function MakeGraphDownwards(root)
30     basicBlock ←GetBasicBlock(root)
31     for inst an instruction after root in basicBlock, going downwards do
32         if inst a memory access or a return instruction then
33             node ←GetNodeBefore(inst)
34             ConnectSink(node)
35             return node
36     node ←GetNodeAtEnd(basicBlock)
37     for basicBlock2 a successor of basicBlock do
38         node2 ←GetNodeAtBeginning(basicBlock2)
39         inst2 ←GetFirstInst(basicBlock2)
40         node3 ←MakeGraphDownwards(inst2)
41         if node3 ≠ NULL then
42             MakeEdge(node, node2)
43             MakeEdge(node2, node3)
```

Algorithm 1: Pseudocode of the fence elimination algorithm



(a) Critical edge in red (b) Breaking the critical edge

Figure 7.2: Breaking critical edges

7.1.3 Corner cases

Function boundaries on the control-flow graph Our program transformation is invoked on each function, and currently does not perform any whole program analysis. In the previous paragraphs we did not detail what to do when the control-flow graph hits the limits of the function being optimised (either the function entry point, or a return instruction). Since a function can be called in an arbitrary context, and a context can perform memory accesses before and after it, the function boundaries are treated as memory accesses (see lines 19 and 32), ensuring that fences at the beginning or end of the function are preserved. Similarly, a function call that may access the memory is treated as if it always modifies the memory. See Section 8.2.2 for ideas on making the algorithm interprocedural.

Critical edges Algorithm 1 does not precisely detail how fences are inserted once the min-cut has been computed. If there is a cut required between the last instruction of a block A and the first instruction of a block B, there are three possibilities:

- block A is the only predecessor of block B: the fence can be inserted at the beginning of block B;
- block B is the only successor of block A: the fence can be inserted at the end of block A;
- if A has several successors, and B has several predecessors, then the edge between A and B is called a *critical edge* since there is no block in which a fence can be inserted ensuring that it is only on the path from A to B. The critical edge is broken by inserting an extra empty basic block between A and B.

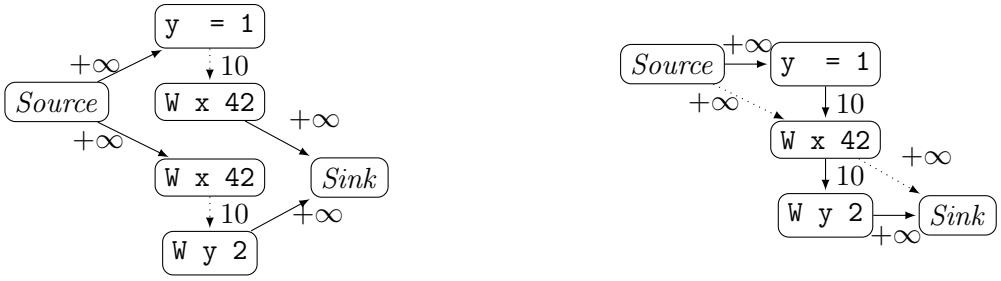
This last case is illustrated in Figure 7.2a, inserting a fence at the location of the red edge is problematic, as inserting it either at A or at B puts it on a path without the red edge. This is solved by breaking the red edge in two, with a block in between, as in Figure 7.2b. The fence can now be safely inserted in block C.

Why two nodes for the each instruction Algorithm 1 systematically creates a node before and a node after each instruction. Perhaps surprisingly, a simpler solution where only one node is created for each instruction would not be be correct. A simple example illustrates this:

```
y = 1;
```

```
fence;
x = 42;
fence;
y = 2;
```

We report the flow graphs for this program, two nodes per memory access on the left, and one node per memory access on the right:



In the flow-graph on the left, the min-cut procedure finds a cut of finite weight (shown by the dotted lines). The edges in this cut are exactly where the fences are in the original program, so the optimisation correctly recognise that the program cannot be optimised.

If instead there is only one node for the store in the middle (flow-graph shown on the right), all cuts between *Source* and *Sink* have an infinite weight, and the min-cut procedure would pick a cut that contains edges involving *Source* and *Sink*. Such edges do not directly correspond to a location in the code, and cannot guide the successive fence-insertion-along-the-cut phase.

7.1.4 Proof of correctness

To show the correctness of our algorithm we first prove a key lemma that states that we may add a fence on a path that had none, but never remove all fences from a path that had at least one. This captures the main intuition that drove our design. The rest of the proof then proceeds by a monotonicity argument. The proof applies to ARM, Power, and x86 instantiations of the "Herding cats" model (Section 2.2.2).

Lemma 11. *For any candidate execution X' of a program P' obtained by applying* `TransformFunction` *to a program P, there is a candidate execution X of P with* $\mathsf{fences}(X) \subseteq \mathsf{fences}(X')$*, and every other part of X is the same as X' (including the events, the* po *and dependency relations, and the* $\xrightarrow{\mathsf{rf}}$ *and* co *relations).*

Proof. Since the algorithm only affects fence placements, we can build X from X' such that it has the same events, po, dependencies, and $\xrightarrow{\mathsf{rf}}$ and co relations. Let a, b be two memory accesses such that $(a, b) \in \mathsf{fences}(X)$. By the Herding Cats construction this implies that there is a path through the control-flow graph that goes first through a, then through a fence f, and finally through b. Since `MakeGraphUpwards` stops at the first memory access encountered (or may stop earlier if it hits the beginning of the function), this path goes through a node connected to the *Source* at some point between a and f. Symmetrically, it goes through a node connected to the *Sink* between f and b. Because `MakeGraphUpwards` and `MakeGraphDownwards` follow the control-flow path, these two nodes must be connected along the path. So the min-cut procedure will have to cut along the path to separate the *Source* from the *Sink*. The algorithm inserts a fence at this point, ensuring $(a, b) \in \mathsf{fences}(X')$. □

This result lifts to executions.

Corollary 2. *For any execution E' of a program P' obtained by applying* `TransformFunction` *to a program P, there is an execution E of P with* $\mathsf{hb}(E) \subseteq \mathsf{hb}(E')$ *and* $\mathsf{prop}(E) \subseteq \mathsf{prop}(E')$*, and every other part of E is the same as E'.*

Proof. The relation fences appears in a positive position in the definition of the hb and prop relations, and every other relation that appears in their definitions (rfe, com, ppo) is invariant by our transformation (since they do not depend on fences, and by Lemma 1 fences are the only part of the program execution that is transformed). □

Given a valid execution of a program, its *final state* is obtained by taking the last write event in the co order for each shared memory location. We state that an optimisation that transform the program P into the program P' is *correct* if for every final state of the memory reached by a valid execution of P', there exists a valid execution of P that ends with the same final state.

Theorem 18. *The program transformation* `TransformFunction` *is correct.*

Proof. We must show that for any valid execution of a program after `TransformFunction`, there is a valid execution of the source program that has the same final state. Corollary 1 provides an effective way to construct an execution E of the source program from an execution E' of the transformed program. Since co is unchanged by the construction and the events are identical, the final state of E and E' is the same. It remains to show that the execution is valid, that is, we must check that the four axioms of the Herding Cats model hold on E. We know that $\mathsf{hb}(E) \subseteq \mathsf{hb}(E')$ and $\mathsf{prop}(E) \subseteq \mathsf{prop}(E')$. We know that the axioms hold on E'. It is straightforward to check that, whenever pairs of elements are added to prop or hb, the four axioms can only be made false, and never true. In turn, since they hold on E' they must also hold on E, ensuring that E is a valid execution of the source program. □

7.2 Implementation and extensions to other ISAs

We have implemented our optimisation as the first optimisation pass in the x86, ARM and IBM Power backends of LLVM 4.0. Min-cut is known to be equivalent to finding a maximum flow in a flow network (via the *max-flow min-cut theorem* of [FF56]), so our implementation uses the standard push-relabel algorithm of [GT88] to compute the min-cut of a graph.

LLVM internally provides the `BlockFrequency` class, that exports a relative metric that represents the number of times a block executes. By default the block frequency info is derived from heuristics, e.g., loop back edges are usually taken, but if profile guided optimisation is enabled then it can rely on runtime profile information. Our implementation accesses the `BlockFrequency` class to compute the weight of the edges (all edges inside a basic block have the weight of the basic block) of the flow graph.

In Section 7.1 we have described the ARM implementation. The IBM Power and x86 passes differ slightly, as detailed below.

Extension to IBM Power As we have seen, the IBM Power memory model is very similar to the ARM one, but is gives to the programmer two different synchronising fence instructions. Heavyweight fences (instruction `hwsync`) are equivalent to the `dmb ish` instruction on ARM, but IBM Power also provides lightweight synchronising fences (instruction `lwsync`) that are faster but offer strictly less guarantees.

We can adapt our algorithm to IBM Power fence instructions by running it in two passes. In a first pass, the heavyweight fences are optimised, while ignoring the lightweight fences. In a second pass, the lightweight fences are optimised, considering every path with a heavyweight fence already cut (since heavyweight fences subsume lightweight ones). This second pass can be implemented by adding the following code snippet after both line 17 and line 35 of Algorithm 1:

```
else if(inst is a heavyweight fence)
  return NULL;
```

This processing in two phases is sound. The first pass preserves the hwsync relation (by the same argument as before). Then the second pass may lose parts of the lwsync relation, but it preserves the fences relation, defined as:

$$\mathsf{fences} \stackrel{\mathsf{def}}{=} (\mathsf{lwsync} \backslash \mathsf{WR}) \cup \mathsf{hwsync}$$

which is the only place where the relation lwsync is used. The proof then proceeds as before based on the monotonicity of the model with respect to both the hwsync and fences relations.

Extension to x86 In the x86 memory model fences prevent write-read reorderings and only matter between stores and loads. This is evident from the axioms: the relation mfence only appears in the definition of hb:

$$\mathsf{hb} = \mathsf{po} \backslash \mathsf{WR} \cup \mathsf{mfence} \cup \mathsf{fre}$$

where po\WR is the program order relation from which every pair from a write to a read has been removed. So we can relax our algorithm without affecting its correctness, and make it preserve WR ∩ mfence rather than mfence. This enables optimising the program on the left into the program on the right (which would have not been possible with the original algorithm):

```
x = 1                x = 1
mfence               x = 2
x = 2                mfence
mfence               tmp = x
tmp = x              tmp = y
mfence
tmp = y
```

To implement this more aggressive optimisation, we alter our algorithm to connect the *Source* to the latest stores before each fence, and connect the earliest loads after them to the *Sink*. Algorithm 1 can be modified by replacing "memory access" by "store" on line 14, and "memory access" by "load" on line 32 of algorithm 1. The proof structure is unaltered: we can still show that for every execution of the optimised program, there is an execution of the original program with hb and prop that are no larger.

7.3 Experimental evaluation

The table in Figure 7.3 considers several concurrent algorithms, including Dekker and Bakery mutual exclusion algorithms, Treiber's stack, as well as a more realistic codebase, LibKPN (See Chapter 6 and [Lê16]). The code of the first three benchmarks is taken from [VZN11], with all shared memory accesses converted to sequentially consistent atomic accesses. The latter is a much larger C11 program (about 3.5k lines) that implements a state-of-the-art dynamic scheduler for lightweight threads communicating through First-In First-Out queues. Atomic qualifiers have been cleverly hand-picked to use the minimum of synchronisation required for correctness.

We report on the number of fences deleted by the x86, ARM, and IBM Power, backend. This is not an ideal metric for our algorithm, because it does not capture cases where a fence is hoisted out of a loop: in this case there is one fence deleted and one inserted, but at run-time a fence per loop iteration might have been optimised away. However it is easily computable and gives some indications on the behaviour of the algorithm.

LLVM's x86 backend does not map sequentially consistent atomic writes to `mov; mfence` but relies on the locked `xchg` instruction, which is believed to be faster. We thus modified the x86 backend to implement an alternative mapping for sequentially consistent accesses: either `mov; mfence` is used for all stores, or `mfence; mov` is used for all loads. We report on both mappings.

The *compile time overhead* due to running our optimisation *is not measurable* and buried in statistical noise, even when compiling larger applications like LibKPN.

Observe that on x86 our algorithm mimics the results of the x86-tailored fence optimisation presented in [VZN11].

For LibKPN we test both the hand-optimised version, and a version where all atomic accesses are replaced by sequentially consistent accesses. On the hand-optimised code, our algorithm still finds opportunities to move and delete a few fences, e.g. reducing it from 25 to 22 on x86, or to move a few around according to the profiling information available. As our optimisation cannot be expressed as a source to source program transformation it is possible that a few fences can be eliminated or moved, even if the qualifier annotations were "optimal". The test where all LibKPN atomic accesses are downgraded to `seq_cst` might suggest that, even though the optimisation pass can eliminate a fair number of redundant fences, the resulting code is not competitive with the hand-tuned implementation.

To investigate this point, and to evaluate the runtime speedup (or slowdown) induced by our of fence optimisation pass, we rely on LibKPN benchmarks. These include both microbenchmarks to test some library constructions, and a larger signal processing application, called `radio`, taken from the *streamIt* benchmark suite[2] and ported to LibKPN. For testing we used a commodity x86 Intel Xeon machine with 12 cores and no hyperthreading, and a 48-core IBM Power7 (IBM 8231-E2C). Unfortunately we do not have an ARM machine available for testing.

Microbenchmarks timings do not benefit from running our optimisations. We conjecture that these stress-test tight loops involving synchronisation and barriers, and as such do not leave much space for improvement of the fence placement. Replacing all the atomic qualifiers with `seq_cst` atomics does not degrade the performance measurably either: this supports our conjecture.

The larger benchmark is much more informative. We report the average and stan-

[2] `http://groups.csail.mit.edu/cag/streamit/`

Benchmark	Configuration	Fences before	after	inserted	deleted
Bakery	ARM	18	16	0	2
Bakery	Power	24	19	10	15
Bakery	x86: all seq-cst, mfence after stores	4	3	0	1
Bakery	x86: all seq-cst, mfence before loads	10	2	1	9
Dekker	ARM	11	9	1	3
Dekker	Power	10	9	5	6
Dekker	x86: all seq-cst, mfence after stores	4	3	0	1
Dekker	x86: all seq-cst, mfence before loads	3	2	2	3
Treiber's stack	ARM	14	13	2	3
Treiber's stack	Power	14	12	4	6
Treiber's stack	x86: all seq-cst, mfence after stores	2	1	0	1
Treiber's stack	x86: all seq-cst, mfence before loads	4	2	2	4
LibKPN	ARM: default	110	110	4	4
LibKPN	ARM: all seq-cst	451	411	11	51
LibKPN	Power: default	92	90	2	4
LibKPN	Power: all seq-cst	448	394	54	108
LibKPN	x86: default	25	22	4	7
LibKPN	x86: all seq-cst, mfence after stores	189	144	5	50
LibKPN	x86: all seq-cst, mfence before loads	326	137	25	214

Figure 7.3: Experimental results (static)

dard deviation (in parentheses) of the performances observed (time in seconds, smaller is better), over 100 invocations of the benchmark:

	x86_64	IBM Power7
original code, not optimised:	252 (67)	1872 (852)
original code, optimised:	248 (55)	1766 (712)
all seq cst, not optimised:	343 (57)	3170 (1024)
all seq cst, optimised:	348 (46)	2701 (892)

The large standard deviation observed is due to the dependance of the benchmark on the scheduling of threads. As a result on x86 performance change due to fence optimisation is hidden in statistical noise, matching the experiments in [VZN11]. Similarly performance improvement due to passing profiling information rather that relying on the default heuristics is also hidden in statistical noise (e.g. we observe 246 (53) for x86_64 on original code, optimised). On Power the results are more encouraging: speedup is observable and it is up to 10% on the code written by a non expert programmer who systematically relies on the sequentially consistent qualifier for atomic accesses.

Chapter 8

Perspectives

The work I presented in this thesis fits in three axes.

Studying the C11 model First, I looked at theoretical properties of the C11 model, and more precisely what compiler optimisations it allows. While doing this, we found several flaws in the standard and proposed fixes to some of them (Chapter 3).

We then proved some criteria of correctness for common compiler optimisations under both the current standard and our proposed fixes (Chapter 4). These criteria are based on local transformations of the traces of memory events, and depend on which fixes are applied to the model.

Improving support of atomics in compilers In a second time, I focused on actual C/C++ compilers and how to improve their support for C11 atomics. As modern C and C++ compilers are very large software projects, made of many different optimisation passes, it is not obvious whether they respect these correctness criteria. And it is even less clear if the criteria change because of issues with the model. So we developed a fuzzy-testing tool that found several bugs in GCC, that could not be found with any other automated tool (Chapter 5).

Following the same goal of improving the support of atomics in real-life compilers, we have proposed, proved correct, and implemented in LLVM, a backend fence optimisation pass (Chapter 7). Our optimisation pass cannot be implemented as a source-to-source transformation in the C11/C++11 memory model, but builds on the precise semantics of the barrier instructions in each targeted architecture (x86, ARM and Power). We have seen that in practice the pass can be effective in removing redundant barriers inserted by a conservative programmer who relies on strong qualifiers (e.g. `seq_cst`) for the atomic accesses.

LibKPN Finally, I worked on LibKPN (Chapter 6), a user-space scheduler for Kahn Process Networks that makes extensive use of C11 atomics. It shows both that C11 atomics are usable in practice, with a few especially useful patterns; and that it is possible to prove algorithms in this model, although in a non modular way.

Artefact availability The formal proofs in Coq of the theorems in Section 4.2 are available from `http://plv.mpi-sws.org/c11comp/`. The fuzzy-testing tool is available from `http://www.di.ens.fr/~zappa/projects/cmmtest/`. Our implementation of the

fence elimination optimisation optimisation, together with the code of the benchmarks, is available from `http://www.di.ens.fr/~zappa/projects/llvmopt`.

8.1 Related work

8.1.1 About optimisations in a concurrent context

As we discussed in Section 4, soundness of optimizations in an idealised DRF model was studied by Ševčík [Sev08, Sev11]. We reuse Ševčík's classification of optimizations as eliminations, reorderings and introductions, but moving from an idealised DRF model to the full C11/C++11 memory model brings new challenges:

- we cannot identify a program with the set of linear orders of actions it can realise because in C and C++ the sequenced-before order is not total; although reasoning about opsems seems unintuitive, partial orders turn out be be easier to work with than the explicit manipulation of trace indices that Ševčík performs;
- the semantics of low-level atomic accesses and fences must be taken into account when computing synchronisations; in particular the weaker consistency and coherency conditions of the release/acquire/relaxed attributes made the soundness proof much more complex.

There are other minor differences, for instance Ševčík assumes that every variable has a default value, while C/C++ forbids accessing an uninitialised variable. Initially Ševčík gave a more restrictive condition for soundness of eliminations [Sev08], namely eliminations are forbidden if there is an intervening release *or* acquire operation rather than a release/acquire pair. This simpler condition appears to be too strong as we have observed compilers eliminate accesses across a release or acquire access. All our analysis was driven by what optimizations we could actually observe: this led to identifying WaW eliminations and RaR/RaW introductions, and motivated us to omit roach-motel reorderings.

There is a long tradition of research on compiler optimizations that preserve sequential consistency dating back to Shasha and Snir [SS88b], mostly focusing on whole program analyses. While these works show that a restricted class of optimizations can maintain the illusion of sequential consistency for all programs, we show that common compiler transformations maintain the illusion of sequential consistency for correctly synchronised programs.

8.1.2 About fuzzy-testing of compilers

Randomised techniques for testing compilers have been popular since the 60's and have been applied to a variety of languages, ranging from Cobol [Sol62] and Fortran [BS96] to C [McK98, Lin05, She07] and C++ [ZXT+09]. A survey (up to 1997) can be found in Boujarwah and Saleh [BS97], while today the state of art is represented by Yang et al.'s work on Csmith [YCER11]. None of these works addresses concurrency compiler bugs and the techniques presented are unable to detect any of our bug reports.

The notable exception is Eide and Regehr's work [ER08] on hunting miscompilation of the `volatile` qualifier in production-quality C compilers. Eide and Regehr generate random well defined *deterministic, sequential* programs with volatile memory accesses:

miscompilations are detected by counting how many accesses to volatile variables are performed during the execution of an unoptimized and an optimized binary of the program. The semantics of the volatile attribute requires that accesses to volatile variables are never optimized away, so comparing the number of runtime accesses is enough to detect bugs in the optimizer. We were inspired by Eide and Regehr's approach to reduce hunting concurrency compilation bugs to analysis to differential testing of sequential code, but the complexity of the C11/C+11 memory model requires a theory of sound optimizations and makes the analysis phase far more complicated.

A paper written after this research was complete used a variant of our method for testing LLVM specifically and found several bugs [CV16]. There are a few differences with the work presented in this chapter:

- They wrote their own program generator instead of using CSmith
- They compare CFGs directly instead of working on traces
- They can check for the LLVM memory model, which differs from the C11 one by allowing write-read races
- They can use compiler instrumentation to help in the matching problem
- The bugs they found are in version 3.6 of LLVM, which was not yet out when we did this research.

8.1.3 About LibKPN

Our runtime lies at the intersection of two research topics: task-based parallelism and dataflow programming.

A large part of task-parallel languages and runtime systems can be classified as discussed in Section 6.1.1, including Cilk [BL99], Swan [VCN], OpenMP [Ope13], SMPSs [PBAL09], Codelet [SZG13], the task engine of Intel TBB, and the *mdf* pattern of FastFlow [ADKT08].

Among lightweight runtime systems, a notable exception is CnC [BBC+10]. It offers a dataflow model with first-class association with synchronization objects and expressivity on par with our proposed model. However, whereas we emphasize reuse, CnC has unbounded and unlimited broadcast on each synchronization object, which prohibits reuse. This is in contrast with SMPSs-style dependencies, which, despite having unbounded broadcast, are not unlimited due to sequential versioning [VCN]. We lack sequential versioning but instead constrain our objects to be uniquely bound. In other words: Swan hyperobjects aggregate multiple dependencies with broadcast, and reuse through sequential versioning and renaming; CnC tags are first-class and support broadcasting, but no reuse; we concentrate on the streaming case, though our objects are first-class yet reusable.

More largely, our work relates to dynamic scheduling in general. Attempts at user-level scheduling can be traced back to m-to-n thread scheduling [ABLL, NM95]. Unlike general m-to-n threading and other user-level threads, all synchronization between tasks in our runtime system is limited to dependencies, with a focus on performance rather than generality (e.g., blocking I/O, system calls, preemption).

Furthermore, our present focus is on dependency tracking and management. Topics such as work queues [FLR, BL99, CL05, LPCZN13], dependence-aware scheduling heuristics [ME], or efficient communication channels [Lam77, GMV08, ADKT08, LBC,

LGCP13], while interesting and certainly useful *in conjunction with* the techniques we describe, are rather orthogonal and not directly related to this work.

The other branch of research, dataflow programming, views programs as graphs of processes communicating through channels (i.e. KPNs [Kah74]). Much prior work in this area has gone into the class of KPNs that are (ultimately) periodic: Synchronous DataFlow (SDF) graphs [LM87] and their derivatives. They have traditionally been the focus of static analysis, optimization and bulk scheduling with barriers [TKA, GTA, KM], as opposed to relying on a runtime scheduler such as ours. Moreover, arbitrary KPNs can be run on top of our runtime, including non periodic and dynamically evolving ones.

8.1.4 About fence elimination

Fence optimisation as a compiler pass When we started this work, the only fence elimination algorithm implemented in LLVM was described in [Elh14]. This is an ARM-specific pass, that proposes both the straightforward elimination of adjacent `dmb ish` instructions in the same basic block with no memory access in between (trivially performed by our algorithm too), and a more involved interblock algorithm. This second algorithm was not integrated into LLVM, making it hard to understand and evaluate. It does not seem to have been proven correct either, and does not take into account profiling information.

Vafeiadis and Zappa Nardelli [VZN11] defined two algorithms (called FE1 and FE2) for removing redundant `mfence` instructions on x86 and proved them correct against the operational description of the x86-TSO memory model. The definition of the optimisations is specific to x86 and the correctness proof leverages complicated simulation-based techniques. Additionally, they proposed a PRE-like step that saturates with `mfence`s all branches of conditionals, in the hope that FE2 will later optimise these away later. Their algorithm does not take into account any profiling information. Our algorithm, once tailored for x86 as described in Section 7.2, subsumes both their optimisation passes, and additionally takes into account profiling information while performing partial redundancy elimination. Our correctness proof, building on an axiomatic model, turns out to be much simpler than theirs.

Fence synthesis based on whole program analysis Several research projects, including [KVY10], [BDM13], [AKNP14], [LNP+12], and [AAC+13], attempt to recover sequential consistency, or weaker safety specifications, by inserting fences in racy programs without atomics annotations. These projects attempt to compute optimal fence insertions but rely on *whole program analyses*: as such these are not directly implementable as optimisation passes in a compiler that performs separate compilation. Also they are too expensive to perform in a general-purpose compiler.

Despite the complementary goals, we remark that the algorithm in [AKNP14] is related to ours: to solve the global optimisation problem it uses an integer linear program solver, of which min-cut is a special case, and correctness follows along similar lines. Our algorithm can thus be seen as a variant of theirs, that trades off optimality for the ability to apply in separate compilation, and to make use of atomics annotation by the programmer. It is interesting to remark that [AKNP14] gives an explicit cost to each fence and lets the solver decide on how to mix them, instead of the heuristic based on profiling information we rely on. Obviously, this is only possible because they do an optimisation based on solving an ILP problem, instead of the simpler min-cut procedure we rely on.

Working inside a compiler, the work by Sura et al. [SFW+05], show how to approximate Sasha and Snir's delay sets and synthesising barriers so that a program has only SC behaviours: they performs much more sophisticated analyses than the ones we propose, but do not come with correctness proofs.

8.2 Future work

8.2.1 About the C11 model

The biggest issue we found in the C11 model is the presence of causality cycles (Section 3.1.2). In addition to the problems we highlighted, they also break attempts at modular verification of C11 programs [BDG13, BD14]. While we proposed some fixes, they all come with serious trade-offs, so the search for a more satisfying solution to this problem is still on, with lots of recent publications [KHL+17, CV17, PS16]. Out of these, I consider [KHL+17] especially remarkable as it seems to fix the causality cycles, while allowing all the optimisations we want to be valid. It achieves this through a complete rewrite of the C11 model to follow an operational model. Repairing the C11 model is especially important, because several other models seem to be looking at borrowing parts of it.

- There are discussions of adding weaker forms of synchronisation to Java (similar to C11 atomic attributes) for performance reasons.While Java memory model is quite different from the C11 one, it is likely to encounter similar problems in the future.
- The Linux kernel currently uses its own macros for atomic accesses, with only informally defined semantics. More primitives have recently been added, including release/acquire accesses reminiscent of C11 for performance reasons. At the same time, some researchers are trying to formalize this memory model to be able to formally prove some parts of the kernel [MAM+16].
- Rust, a new language developed at Mozilla, uses the relaxed memory operations in the style of C11 [Rus]. So any fix to the C11 model would also benefit Rust users.
- Finally, the LLVM compiler used to claim that the memory model of its middle-end IR is the same as C11. It has been recently found that it is slightly different, and possibly better [CV17].

In terms of compiler optimisations of memory accesses, some of those we proved unsound in some models might be sound under some additional constraint. The most obvious such constraint is that the affected memory-location be local memory. I conjecture that all common optimisations of non-atomic accesses are sound in this context.

8.2.2 About the fence elimination algorithm

Speaking of LLVM, it would be interesting to further investigate experimentally in which cases there is a benefit in replacing the `xchg` mapping for sequentially consistent stores used by LLVM with the `mov; mfence` mapping backed by an aggressive fence optimisation pass. More generally, large benchmarks to test performance of concurrent C11/C++11 code bases with our fence elimination algorithm are still missing, and highly needed to perform a more in-depth performance evaluation.

Our algorithm can be improved in several ways, detailed below.

Interprocedural analysis As mentioned in Section 7.1.3, our algorithm is intraprocedural and treats any function call that may touch memory as a memory access. It is possible to slightly improve this by building function dictionaries. For this, starting from the leaves of the call-graph, we can annotate each function with:

- whether all paths from the first instruction hit a fence before a memory access or return instruction;
- whether all paths flowing backwards from a return instruction hit a fence before a memory access or the starting instruction.

Then, anytime a function call is encountered while looking for an access after (resp. before) a fence, and the first (resp. second) flag is set on that function, that path can be cut. I believe this improvement is probably the one with the highest impact.

Leveraging coherency It is possible to implement a more aggressive optimisation if the algorithm keeps track of the C11/C++11 atomic access responsible for inserting each fence instruction. Consider a release store to a variable `x` on ARM: this is compiled to a `dmb ish` barrier followed by the store instruction. Since the semantics of release forces only to order previous accesses with this store and not with every later access, and accesses to the same location are already ordered locally (by the *SC-per-loc* rule of the Herding Cats framework), it is possible to ignore all accesses to `x` when looking for memory accesses to connect the *Source* node to.

The same idea can be applied to acquire or sequentially consistent loads. In particular, this would allow sinking the fence out of the loop when a loop contains only an acquire load, having the effect of transforming:

```
while (!x.load(acquire)) {};
```

into:

```
while(!x.load(relaxed)) {};
fence(acquire);
```

This might be more efficient, but would need benchmarking. It is also not clear at this time if there are other common patterns that could be optimised in this way.

Optimising for code size Following an idea from [SHK04], it is possible to optimise for code size instead of runtime. Our algorithm can be easily adapted by using the constant 1 for the weight of all edges, instead of the frequency in the profiling information. The min-cut procedure will then minimise the number of fences in the generated code. A tradeoff between code-size and runtime can easily be obtained by using a hybrid metric (this idea comes from [SHK04]).

A conservative variant of the algorithm As presented above, our algorithm is fundamentally speculative: it relies on the profiling information and can significantly worsen the performance of the code if this information is unreliable.

However any path that goes through a location dominated or post-dominated by a fence necessarily goes through that fence (by definition of dominators and post-dominators). Consequently, it is safe to put fences in all of these locations: there is

no risk of introducing a fence on a path that had none. We conjecture that we can get a conservative variant of our algorithm by connecting the first node (resp. the last node) of any block in `MakeGraphUpwards` (resp. `MakeGraphDownwards`) that has at least one predecessor (resp. successor) that is not post-dominated (resp. dominated) by the original fence to the source (resp. the sink).

The paper [XK06] proposes a better solution to get a conservative algorithm, relying on dataflow analyses. Unfortunately this approach is hard to integrate with our currently implemented algorithm.

8.2.3 About LibKPN

The part of the thesis I would most like to expand in the next few months is LibKPN (Chapter 6). While there is already a complete implementation of algorithms F and S, and some very promising benchmarks, the algorithm H has not been fully implemented yet. I also hope to build some high level API/DSL for describing KPNs, as the current interface is extremely low-level and consequently error-prone.

Bibliography

[AAC+13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 530–536, 2013.

[ABLL] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *SOSP '91.*

[ADKT08] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, March 2008.

[AFI+09] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24, 2009.

[AGH+11] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[AKNP14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 508–524, 2014.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 141–157, 2013.

[alp98] Alpha architecture handbook, 1998. `http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=alphaahb.pdf`.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 258–272, 2010.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 7, 2014.

[BA08] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[BBC+10] Zoran Budimlič, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Taşirlar. Concurrent collections. *Sci. Program.*, 18:203—217, 2010.

[BD14] Hans-Juergen Boehm and Brian Demsky. Outlawing ghosts: avoiding out-of-thin-air results. In *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC '14, Edinburgh, United Kingdom, June 13, 2014*, pages 7:1–7:6, 2014.

[BDG13] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 235–248, New York, NY, USA, 2013. ACM.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 533–553, Berlin, Heidelberg, 2013. Springer-Verlag.

[BDW16] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and opencl. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 634–648, 2016.

[Bec11] Pete Becker. *Standard for Programming Language C++ - ISO/IEC 14882*, 2011.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 1999.

[BLKD09] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[BMO+12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66, 2011.

[BP09] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 392–403, 2009.

[BS96] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information & Software Technology*, 38(2):111–119, 1996.

[BS97] Abdulazeez S. Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9):617–625, 1997.

[C11] C/C++11 mappings to processors.

[CDO+96] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program.*, 5(3), 1996.

[CL05] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 21–28, 2005.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

[CV16] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*, pages 216–226, 2016.

[CV17] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *International Symposium on Code Generation and Optimization (CGO)*, 2017.

[DPS10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, pages 185–192, Washington, DC, USA, 2010. IEEE Computer Society.

[Elh14] Reinoud Elhorst. Lowering c11 atomics for arm in llvm. 2014.

[ER08] Eric Eide and John Regehr. Volatiles are miscompiled and what to do about it. *EMSOFT*, 2008.

[FF56] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[FLR] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI'98*.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, New York, NY, 2008. ACM.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[GTA] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs.

[HH97] R Nigel Horspool and HC Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *Computer Systems and Software Engineering, 1997., Proceedings of the Eighth Israeli Conference on*, pages 111–118, 1997.

[ita02] Intel® itanium® architecture software developer's manual, 2002. `http://www.ece.lsu.edu/ee4720/doc/itanium-arch-2.1-v2.pdf`.

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[KHL+17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189, 2017.

[KM] Manjunath Kudlur and Scott Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. PLDI'08.

[KVY10] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010. FMCAD Inc.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, SE-3(2):125–143, 1977.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[LBC] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *ANCS'09*.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[Lê16] Nhat Minh Lê. *Kahn process networks as concurrent data structures: lock freedom, parallelism, relaxation in shared memory*. PhD thesis, École normale supérieure, France, December 2016.

[LGCP13] Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and efficient bounded FIFO queues. In *25th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2013, Porto de Galinhas, Pernambuco, Brazil, October 23-26, 2013*, pages 144–151, 2013.

[Lin05] Christian Lindig. Random testing of C calling conventions. In *AADEBUG*, 2005.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[LNP+12] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 429–440, New York, NY, USA, 2012. ACM.

[LPCZN13] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 69–80, 2013.

[LV15] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, ICALP 2015, pages 311–323, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[MAM+16] Paul E. McKenney, Jade Alglave, Luc Maranget, Andrea Parri, and Alan Stern. Linux-kernel memory ordering: Help arrives at last!,

2016. http://www2.rdrop.com/~paulmck/scalability/paper/LinuxMM.2016.10.26c.LPC.pdf.

[McK98] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[McK16] Paul McKenney. P0190r0: Proposal for new memory_order_consume definition, 2016.

[ME] Changwoo Min and Young Ik Eom. DANBI: Dynamic Scheduling of Irregular Stream Programs for Many-core Systems. In *PACT'13*.

[MMS+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 495–512, 2012.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[MPZN13] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 187–196, 2013.

[MS11] P. E. McKenney and R. Silvera, 2011. http://www.rdrop.com/users/paulmck/scalability/!paper/N2745r.2011.03.04a.html.

[MZN17] Robin Morisset and Francesco Zappa Nardelli. Partially redundant fence elimination for x86, ARM, and Power processors. In *Compiler Construction*, 2017.

[ND13] Brian Norris and Brian Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 131–150, New York, NY, USA, 2013. ACM.

[NM95] Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures. In *PARCO*, 1995.

[Ope13] The OpenMP 4.0 specification, July 2013.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.

[PBAL09] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming With StarSs. *Int. J. on High Performance Computing Architecture*, 23(3), 2009.

[PC13] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.

[PS16] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 622–633, 2016.

[RCC+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.

[Rus] Rust atomics documentation. `https://doc.rust-lang.org/core/sync/atomic/`.

[SA08] Jaroslav Sevcík and David Aspinall. On validity of program transformations in the java memory model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 27–51, 2008.

[Sev08] Jaroslav Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[Sev11] Jaroslav Sevcik. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 2–13, New York, NY, USA, 2005. ACM.

[She07] Flash Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[SHK04] Bernhard Scholz, R. Nigel Horspool, and Jens Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, USA, June 11-13, 2004*, pages 221–230, 2004.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322, 2012.

[Sol62] R. L. Solder. A general test data generator for COBOL. In *AFIPS Joint Computer Conferences*, 1962.

[SS88a] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[SS88b] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2), 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[SZG13] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. An implementation of the codelet model. In *Euro-Par*, pages 633–644, 2013.

[TA] William Thies and Saman P. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT'10*.

[Ter08] A. Terekhov. Brief tentative example x86 implementation for C/C++ memory model, 2008. `http://www.decadent.org.uk/pipermail/~cpp-threads/2008-December/001933.html`.

[TKA] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC'02*.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 691–707, 2014.

[VBC+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220, 2015.

[VCN] Hans Vandierendonck, Kallia Chronaki, and Dimitrios S. Nikolopoulos. Deterministic scale-free pipeline parallelism with hyperqueues. In *SC'13*.

[VN13] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 867–884, 2013.

[VZN11] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 146–162, 2011.

[WG03] David L. Weaver and Tom Germond. The sparc architecture manual, 2003. `http://pages.cs.wisc.edu/~fischer/cs701.f08/sparc.v9.pdf`.

[XK06] Jingling Xue and Jens Knoop. A fresh look at PRE as a maximum flow problem. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, pages 139–154, 2006.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[ZXT+09] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *AST*, 2009.

Appendix A

Proof of Theorem 4

Construction A.0.1. From a consistent execution (O', W') of a program P' which is an elimination of a well defined program P we want to build a candidate execution (O, W). First, we pick an opsem $O \in P$ such that O' is an elimination of O. Then we build a related witness W by taking W', and computing $<_{\mathsf{hb}}$. Finally we modify W as follows:

- RaR: If i is a RaR justified by a read r, then for every write w such that $w <_{\mathsf{rf}} r$, we add $w <_{\mathsf{rf}} i$
- RaW: If i is a RaW justified by a write w, then we add $w <_{\mathsf{rf}} i$
- IR: If i is a IR, then if there is a write w to the same location with $w <_{\mathsf{hb}} i$ we pick an opsem O such that i reads the same value, and add $w <_{\mathsf{rf}} i$ to W.
- OW: No change to W
- WaW: If a is a WaW justified by a write w, then for every read r such that $w <_{\mathsf{rf}} r$ and $a <_{\mathsf{hb}} r$ we replace $w <_{\mathsf{rf}} r$ by $a <_{\mathsf{rf}} r$
- WaR: If a is a WaR, then for every read r of the same value at the same location such that $a <_{\mathsf{hb}} r$ and either $w <_{\mathsf{rf}} r$ with $w <_{\mathsf{hb}} a$ or r reads from no write, we add $a <_{\mathsf{rf}} r$. (replacing $w <_{\mathsf{rf}} r$ if it exists)

Lemma A.0.1. *If (O', W') is a consistent execution, then so is (O, W)*

Proof. As we preserve everything but $<_{\mathsf{rf}}$ from W', consistent non atomic read values is the only predicate that is hard to prove. It is wrong if there is w_1 and w_2 two non atomic writes at the same location, and i a non atomic read at the same location with $w_1 <_{\mathsf{hb}} w_2 <_{\mathsf{hb}} i$ and $w_1 <_{\mathsf{rf}} i$. We check every case below to prove that this is impossible.

- RaR:
  If i is not the RaR it is impossible as exactly the same pattern would occur in (O', W'). So we assume that i is a RaR justified by a read r. Because of how the construction works, $w_1 <_{\mathsf{rf}} r$. So because (O', W') is a candidate execution, $w_1 <_{\mathsf{hb}} r$. Three cases must be examined:
  - w_2 and r are not comparable by $<_{\mathsf{hb}}$:
    We use the prefix-closedness of opsemsets to find an opsem $\tilde{O} \in P$ that is a prefix of O that does include w_2 and r but not i. As it does not contain i, the restriction of W to it makes it a candidate execution and it exhibits a data-race between w_2 and r. Absurd as P is well defined.

- $w_2 <_{\mathsf{hb}} r$:
  Then $w_1 <_{\mathsf{hb}} w_2 <_{\mathsf{hb}} r$ and $w_1 <_{\mathsf{rf}} r$ in (O', W') as well which is impossible as it is a candidate execution.
- $r <_{\mathsf{hb}} w_2$:
  If w_2 is in the same thread as r and i, then $r <_{\mathsf{sb}} w_2 <_{\mathsf{sb}} i$, which is impossible by definition of RaR. Otherwise, there must be a release action a, and an acquire action b such that $r <_{\mathsf{sb}} a <_{\mathsf{hb}} w_2 <_{\mathsf{hb}} b <_{\mathsf{sb}} i$, which is also impossible by definition of RaR

- RaW:
  Again, if the eliminated access is not i, the same problem would happen in (O', W'), so we assume i is the eliminated access. So there must be a write w that justifies i being a RaW and $w <_{\mathsf{rf}} i$. By construction we only add one $<_{\mathsf{rf}}$ edge, so $w = w_1$ and $w_1 <_{\mathsf{sb}} i$. w_2 can't be in the same thread by definition of RaW and if it is in another thread, there must be a release-acquire pair between w_1 and r to justify the $<_{\mathsf{hb}}$ edges, which is again impossible.

- IR:
  Impossible by construction, as we chose a write that is maximal with regards to $<_{\mathsf{hb}}$ as the origin of the $<_{\mathsf{rf}}$ edge.

- OW:
  Because in this case we do not change any $<_{\mathsf{rf}}$ edge, the only hard case (for which the problem is not directly present in (O', W'), is if w_2 is the OW. So there is a write w_3 that justifies it, and $w_2 <_{\mathsf{sb}} w_3$. There are three cases to be looked at:

  - w_3 and i are not comparable by $<_{\mathsf{hb}}$:
    By receptiveness of opsemsets we can pick a new one where i reads the same value as w_2. Then we use prefix-closedness to get a prefix $\tilde{O}$ from that opsem, that contains w_3 and i but nothing after i. By restricting W to that opsem, and replacing $w_1 <_{\mathsf{rf}} i$ by $w_2 <_{\mathsf{rf}} i$, we get a candidate execution. It has a data-race (or an unsequenced-race) between w_3 and i, which is impossible as P is well defined.
  - $w_3 <_{\mathsf{hb}} i$:
    Then we also have $w_1 <_{\mathsf{hb}} w_3 <_{\mathsf{hb}} i$ and $w_1 <_{\mathsf{rf}} i$ in (O', W') which is impossible as (O', W') is a candidate execution
  - $i <_{\mathsf{hb}} w_3$:
    Either i is in the same thread (impossible by definition of OW) or there is a release-acquire pair between w_2 and w_3 to explain the $<_{\mathsf{hb}}$ edges (impossible for the same reason)

- WaW:
  Either of the writes can be the eliminated one:

  - If w_1 is the WaW:
    It is justified by a write w, with $w <_{\mathsf{hb}} w_2 <_{\mathsf{hb}} i$ and $w <_{\mathsf{rf}} i$ in (O', W'), which is impossible as (O', W') is a candidate execution.
  - If w_2 is the WaW, justified by a write w:

* If w and w_1 are not comparable by $<_{\mathsf{hb}}$:
  We use the prefix-closedness of opsemsets to find an opsem $\tilde{O} \in P$ that is a prefix of O that does include w and w_1 but not w_2 or i. As it does not contain i, the restriction of W to it makes it a candidate execution and it exhibits a data-race between w and w_1. Absurd as P is well defined.
* If $w <_{\mathsf{hb}} w_1$:
  Either w_1 is in the same thread as w and w_2 or there is a release-acquire pair between them to account for the $<_{\mathsf{hb}}$ edges. Both are impossible by the definition of WaW
* If $w_1 <_{\mathsf{hb}} w$:
  Then $w_1 <_{\mathsf{hb}} w <_{\mathsf{hb}} i$ and $w_1 <_{\mathsf{rf}} i$ in (O', W') which is impossible because it is a candidate execution.

- WaR:
  - If w_1 is the WaR:
    In (O', W'), $w_2 <_{\mathsf{hb}} i$, so i must be from some write w_0 as (O', W') is a candidate execution. By construction, $w_0 <_{\mathsf{hb}} w_1$ So $w_0 <_{\mathsf{hb}} w_2 <_{\mathsf{hb}} i$ and $w_0 <_{\mathsf{rf}} i$ in (O', W'), which is impossible as it is a candidate execution.
  - If w_2 is the WaR, justified by a read r:
    * If w_1 and r are not comparable by $<_{\mathsf{hb}}$:
      We use the prefix-closedness of opsemsets to find an opsem $\tilde{O} \in P$ that is a prefix of O that does include w_1 and r but not w_2 or i. As it does not contain i, the restriction of W to it makes it a candidate execution and it exhibits a data-race between w_1 and r. Absurd as P is well defined.
    * If $w_1 <_{\mathsf{hb}} r$:
      By construction, there must exist some write w_3 such that $w_3 <_{\mathsf{rf}} r$. Four cases follow:
      · w_3 and w_1 are not comparable by $<_{\mathsf{hb}}$:
        We use the prefix-closedness of opsemsets to find an opsem $\tilde{O} \in P$ that is a prefix of O that does include w_3 and w_1 but not w_2 or i. As it does not contain i, the restriction of W to it makes it a candidate execution and it exhibits a data-race between w_3 and w_1. Absurd as P is well defined.
      · $w_3 <_{\mathsf{hb}} w_1$:
        We would have $w_3 <_{\mathsf{hb}} w_1 <_{\mathsf{hb}} r$ and $w_3 <_{\mathsf{rf}} r$ in (O', W'), which is impossible as it is a candidate execution
      · $w_1 <_{\mathsf{hb}} w_3$:
        We would have $w_1 <_{\mathsf{hb}} w_3 <_{\mathsf{hb}} i$ and $w_1 <_{\mathsf{rf}} i$ in (O', W'), which is impossible as it is a candidate execution
      · $w_1 = w_3$:
        As (O', W') is a candidate execution, and $w_1 <_{\mathsf{rf}} r$ in it, w_1 is of the same value as r, and thus of the same value as w_2. So by construction i would read from w_2 instead of w_1.

□

Lemma A.0.2. *If (O', W') a candidate execution exhibits an undefined behavior, then so does (O, W) (built by the construction above).*

Proof. We look at the the four possible cases:

- Unsequenced race:
  If a and b are in an unsequenced race in (O', W'), they still are in it in (O, W) as we do not change $<_{\mathsf{sb}}$ in the construction

- Data race:
  The same: $<_{\mathsf{hb}}$ is preserved during the construction

- Indeterminate read:
  We mostly add $<_{\mathsf{rf}}$ edges in the construction to new reads, or we replace already-existing edges. The only exception is in the case for WaR: If w is a WaR justified by a read r, i is an indeterminate read and $w <_{\mathsf{hb}} i$, then $w <_{\mathsf{rf}} i$ in (O, W) and not in (O', W'). However, if there exists w_1 such that $w_1 <_{\mathsf{rf}} i$ in (O', W'), then $w_1 <_{\mathsf{hb}} r$, as (O', W') is a candidate execution. And thus $w_1 <_{\mathsf{hb}} i$ as $r <_{\mathsf{sb}} w <_{\mathsf{hb}} i$, which is impossible as $(O, W)'$ is a candidate execution. So there is no such w_1, and r is also an indeterminate read, in both (O', W') and (O, W)

- Bad mutexes:
  Trivial as we do not affect synchronisation actions in any way.

□

Theorem A.0.1. *Let the opsemset P' be an elimination of an opsemset P. If P is well defined, then so is P' and any execution of P' has the same behavior as some execution of P.*

Proof. First, if P' were to be ill-defined, there would be a candidate execution (O', W') of P' that exhibits an undefined behaviour. By the above lemmata, it is possible to build a corresponding candidate execution (O, W) of P that also exhibits an undefined behaviour, which is impossible as P is assumed to be well defined. So P' is well defined
And by the lemmata above, for every execution (O', W') of P' we can build an execution (O, W) of P, that have the exact same observable behavior (by construction, we do not affect in any way synchronisation actions). □

Résumé

Les architectures modernes avec des processeurs multicoeurs, ainsi que les languages de programmation modernes, ont des mémoires faiblement consistentes. Leur comportement est formalisé par le modèle mémoire de l'architecture ou du langage de programmation; il définit précisément quelle valeur peut être lue par chaque lecture dans la mémoire partagée. Ce n'est pas toujours celle écrite par la dernière écriture dans la même variable, à cause d'optimisation dans les processeurs, telle que l'exécution spéculative d'instructions, des effets complexes des caches, et des optimisations dans les compilateurs.

Dans cette thèse, nous nous concentrons sur le modèle mémoire C11 qui est défini par l'édition 2011 du standard C. Nos contributions suivent trois axes.

Tout d'abord, nous avons regardé la théorie autour du modèle C11, étudiant de façon formelle quelles optimisations il autorise les compilateurs à faire. Nous montrons que de nombreuses optimisations courantes sont permises, mais, surprenamment, d'autres, importantes, sont interdites.

Dans un second temps, nous avons développé une méthode à base de tests aléatoires pour détecter quand des compilateurs largement utilisés tels que GCC et Clang réalisent des optimisations invalides dans le modèle mémoire C11. Nous avons trouvés plusieurs bugs dans GCC, qui furent tous rapidement fixés. Nous avons aussi implémenté une nouvelle passez d'optimisation dans LLVM, qui recherchent des instructions des instructions spéciales qui limitent les optimisations faites par le processeur - appelées instructions barrières - et élimine celles qui ne sont pas utiles.

Finalement, nous avons développé un ordonnanceur en mode utilisateur pour des threads légers communicants via des canaux premier entré-premier sorti à un seul producteur et un seul consommateur. Ce modèle de programmation est connu sous le nom de réseau de Kahn, et nous montrons comment l'implémenter efficacement, via les primitives de synchronisation de C11. Ceci démontre qu'en dépit de ses problèmes, C11 peut être utilisé en pratique.

Mots Clés

Modèle mémoire, C11, Optimisations, Compilateur, Barrière.

Abstract

Modern multiprocessors architectures and programming languages exhibit weakly consistent memories. Their behaviour is formalised by the memory model of the architecture or programming language; it precisely defines which write operation can be returned by each shared memory read. This is not always the latest store to the same variable, because of optimisations in the processors such as speculative execution of instructions, the complex effects of caches, and optimisations in the compilers.

In this thesis we focus on the C11 memory model that is defined by the 2011 edition of the C standard. Our contributions are threefold.

First, we focused on the theory surrounding the C11 model, formally studying which compiler optimisations it enables. We show that many common compiler optimisations are allowed, but, surprisingly, some important ones are forbidden.

Secondly, building on our results, we developed a random testing methodology for detecting when mainstream compilers such as GCC or Clang perform an incorrect optimisation with respect to the memory model. We found several bugs in GCC, all promptly fixed. We also implemented a novel optimisation pass in LLVM, that looks for special instructions that restrict processor optimisations - called fence instructions - and eliminates the redundant ones.

Finally, we developed a user-level scheduler for lightweight threads communicating through first-in first-out single-producer single-consumer queues. This programming model is known as Kahn process networks, and we show how to efficiently implement it, using C11 synchronisation primitives. This shows that despite its flaws, C11 can be usable in practice.

Keywords

Memory model, Memory consistency model, C11, Compiler optimisations, Fence.