Low-level Concurrent Programming Using the Relaxed Memory Calculus

Michael J. Sullivan

CMU-CS-17-126

November 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Karl Crary, Chair
Kayvon Fatahalian
Todd Mowry
Paul McKenney, IBM

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored in part by the Carnegie Mellon University Presidential Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: programming languages, compilers, concurrency, memory models, relaxed memory

Abstract

The Relaxed Memory Calculus (RMC) is a novel approach for portable low-level concurrent programming in the presence of the the relaxed memory behavior caused by modern hardware architectures and optimizing compilers. RMC takes a declarative approach to programming with relaxed memory: programmers explicitly specify constraints on execution order and on the visibility of writes. This differs from other low-level programming language memory models, which—when they exist—are usually based on ordering annotations attached to synchronization operations and/or explicit memory barriers.

In this thesis, we argue that this declarative approach based on explicit programmer-specified constraints is a practical approach for implementing low-level concurrent algorithms. To this end, we present RMC-C++, an extension of C++ with RMC constraints, and `rmc-compiler`, an LLVM based compiler for it.

We live on a placid island of ignorance in the midst of black seas of infinity,
and it was not meant that we should voyage far.
— “The Call of Cthulhu”, H.P. Lovecraft

And all I ask is a tall ship and a star to steer her by;
— “Sea Fever”, John Masefield

Acknowledgments

It's difficult to overstate how much thanks is owed to my parents, Jack and Mary Sullivan. They've provided nearly three decades of consistent encouragement, support (emotional, moral, logistical, financial, and more), and advice ("Put 'write thesis proposal' on your To-Do List, then write the proposal and cross it off!"). I know I wasn't the easiest kid to raise, but they managed to mostly salvage the situation. Thank you—I wouldn't be here without you.

Thanks to my brother, Joey, who my interactions with over the years have shaped me in more ways than I understand. And I'm very grateful that—at this point in our lives—most of those interactions are now positive.

Immeasurable thanks to my advisor, Karl Crary, without whom this never would have happened. Karl is an incredible researcher and it was a pleasure to spend the last six years collaborating with him. Karl also—when I asked him about the CMU 5th Year Masters—was the one to convince me to apply to Ph.D. programs instead. Whether *that* is quite worth a "thanks" is a hard question to answer with anything approaching certainty, but I believe it is.

I also owe a great debt to Dave Eckhardt. Operating Systems Design and Implementation (15-410) is the best class that I have ever taken and I haven't been able to get it out of my head. The work I did over seven semesters as one of Dave's TAs for 410—teaching students about concurrency, operating systems, and, most of all, giving thoughtful and detailed feedback about software architecture and design—is the work I am most proud of, and I am supremely thankful for being able to be a part of it. While much of my work is likely to take me far afield, I try to carry a little of the kernel hacker ethic in my heart wherever I go. On top of that, Dave has been a consistent source of advice, support, and guidance without which I'd never have made it through grad school.

I owe much to a long string of excellent teachers I had in the Mequon-Thiensville School District, beginning with my third-grade teacher Mrs. Movall, who—in response to me asking if there was a less tedious way to draw some particular shape in LOGO—arranged for a math teacher to come over from the high school to teach me about the wonders of variables, if statements, and GOTOs. I owe a particularly large debt to Robin Schlei, who provided years of support, and to Kathleen Connelly, who first put me in touch with Mark Stehlik. Thank you!

On that note, thanks to Mark Stehlik, who was as good an undergraduate advisor as could be asked for, even if he wouldn't let me count HOT Compilation as an "applications" class. And thanks to all the great teachers I had at CMU, especially Bob Harper, who kindled an interest in programming languages.

Grad school was a long and sometimes painful road, and I'd have never made it through without my hobbies. Chief among them, my regular tabletop role-playing game group, who sacrificed countless Saturday nights to join me in liberating Polonia, exploring the Ironian Wastes for the Ambrosia Trading Company, and conclusively demonstrating, via their work for the Kromian Mages Guild, that not all wizards are subtle: Rhett Lauffenburger (Roger Kazynski), Gabe Routh (Hrothgar Boarbuggerer né Hrunting), Ben Blum (Yolanda Swaggins-Moonflute), Matt Maurer (Jack Silver), Michael Arntzenius (Jakub the Half-Giant), and Will Hagen (Daema). A GM couldn't ask for a better group of players. I'd also like to thank my other major escape: the EVE Online alliance Of Sound Mind, and especially the CEOs who have built SOUND into such a great group to fly with—XavierVE, June Ting, Ronnie Cordova, and Jacob Matthew Jansen.

Fly safe!

The SCS Musical provided another consistently valuable form of stress in my life, and I'd like to thank everybody involved in helping to revive it and keep it running. I had allowed myself to forget about the giant technical-theatre-shaped hole in my heart, and I don't think I'll be able to again.

Thanks to the old gang from Homestead—Becki Johnson, Gina Buzzanca, and Michael and Andrew Bartlein. Though we've been scattered far and wide from our parents' basements in Wisconsin—and though the Bartleins are rubbish at keeping in touch—you've remained some of my truest friends.

A lot of people provided—in their own ways—important emotional support at different points in the process. I owe particular thanks to Jack Ferris, Joshua Keller, Margaret Meyerhofer, Amanda Watson, and Sarah Schultz.

Salil Joshi and Joe Tassarotti contributed in important ways to the design of RMC and its compiler. Ben Segall, Paul Dagnelie, and Rick Benua provided helpful suggestions about benchmarking woes. Paul and Rick also provided useful comments on the draft. Matt Maurer provided some much needed statistical help. I'd like to thank Shaked Flur et al. for providing me with the ARMv8 version of the ppcmem tool, which proved invaluable for exploring the possible behaviors of ARMv8s.

A heartfelt thanks to everyone on my committee. Paul McKenney provided early and interesting feedback on this document (and a quote from "Sea Fever"), Kayvon Fatahalian gave excellent advice on what he saw as the key arguments of my work, and Todd Mowry helped me understand the computer architect's view of the world a little better.

I've had a lot of fantastic colleagues during the decade I spent at CMU, both as an undergrad and as a grad student. I learned more about computer science and about programming languages in conversation with my colleagues than I have from any books or papers. Thanks to Joshua Wise, Jacob Potter, Ben Blum, Ben Segall, Glenn Willen, Nathaniel Wesley Filardo, Matthew Wright, Elly Fong-Jones, Philip Gianfortoni, Car Bauer, Laura Abbott, Chris Lu, Eric Faust, Andrew Drake, Amanda Watson, Robert Marsh, Jack Ferris, Joshua Keller, Joshua Watzman, Cassandra Sparks, Ryan Pearl, Kelly Hope Harrington, Margaret Meyerhofer, Rick Benua, Paul Dagnelie, Michael Arntzenius, Carlo Anguili, Matthew Maurer, Joe Tassarotti, Rob Simmons, Chris Martens, Nico Feltman, Favonia, Danny Gratzer, Stefan Mueller, Anna Gommerstadt, and many many others.

Thanks also to all the wonderful colleagues I had working at Mozilla and Facebook on the Rust and Hack programming languages.

The most stressful and miserable part of my grad school process was the semester I spent preparing my thesis proposal. I was nearly burnt out after it and I don't think I could have kept on going without the ten weeks I spent "Working From Tahoe" (WFT) from The Fairway House in Lake Tahoe. Thanks to Vail Resorts and everybody who split the ski cabin with me for making it possible—especially to Jack Ferris and Kelly Hope Harrington for bullying me into it. And thanks to the Northstar Ski Patrol, the paramedics of the Truckee Fire Protection District, and the staff at the Tahoe Forest Hospital and the University of Pittsburgh Medical Center for all the excellent care I received after I skied into a tree.

Thanks to Aaron Rodgers and Mike McCarthy, but not to Dom Capers. Go Pack Go!

NVIDIA Corporation donated a Jetson TK1 development kit which was used for testing

and benchmarking. The IBM Power Systems Academic Initiative team provided access to a POWER8 machine which was used for testing and benchmarking.

To all of the undoubtedly countless people who I really really ought to have thanked here but who slipped my mind: I'm sorry, and thank you!

And last, but certainly not least, I'd like to extend a heartfelt thanks to Phil, Madz, Manny, and all the rest of the gang down at Emarhavil Heavy Industries. Emarhavil Heavy Industries—where there's always something big just over the horizon.

Contents

Chapter 1

Introduction

Writing programs with shared memory concurrency is notoriously difficult even under the best of circumstances. By "the best of circumstances", we mean something specific: when memory accesses are sequentially consistent. Sequential consistency promises that threads can be viewed as strictly interleaving accesses to a single shared memory [31]. Unfortunately, sequential consistency can be violated by CPU out-of-order execution and memory subsystems as well as by many very standard compiler optimizations.

Traditionally, languages approach this by guaranteeing that data-race-free code will behave in a sequentially consistent manner. Programmers can then use locks and other techniques to synchronize between threads and rule out data races. However, for performance-critical code and library implementation this may not be good enough, requiring languages that target these domains to provide a well defined low-level mechanism for shared memory concurrency. C and C++ (since the C++11 and C11 standards) provide a mechanism based around specifying "memory orderings" when accessing concurrently modified locations. These memory orderings induce constraints that limit the behavior of programs. The definitions here are very complicated, though, with lots of moving parts.

We propose a new, more declarative approach to handling weak memory in low-level concurrent programming based on the Relaxed Memory Calculus (RMC) [18]: explicit, programmer specified constraints. In RMC, the programmer explicitly specifies constraints on the order of execution of operations and on the visibility of memory writes.

1.1 Sequential Consistency

Sequential consistency is the gold standard for compiling and executing concurrent programs that share memory [31]. An execution of a concurrent program is sequentially consistent if it is equivalent to executing some interleaving of execution of instructions from different threads, all accessing a single shared memory that maps from addresses to values. That is, threads run their code in order, sharing a single memory. While writing and reasoning about concurrent programs can still be quite difficult (because there can be many possible interleavings!), the model is easy to understand and much work has been put into developing tools and techniques for reasoning about it.

1.2 Paradise Lost

1.2.1 Hardware architecture problems

Modern multi-core architectures almost universally do not provide sequential consistency. The guarantees provided by different architectures, and the complexity of their respective models, can vary substantially. Intel's x86 architecture is very tame in the behaviors it allows and can be modeled in a fairly straightforward manner by considering each CPU to have a FIFO store buffer of writes that have not yet been propagated to the main memory [42]. Other architectures, such as POWER and ARM, have memory models so relaxed that most models [2, 40] dispense with the pleasant fiction of "memory" (a mapping from addresses to values) and instead work with a set of memory operations related by various partial orders.

One of the most common relaxed behaviors (and really the only one permitted on x86) is *store buffering*:

```
*p = 1;    *q = 1;
r1 = *q;   r2 = *p;
```

Is `r1 == r2 == 0` allowed?

Here, two threads each write to a different shared variable, and then read from the variable that the other thread wrote to (this pattern is at the heart of some important theoretical mutual exclusion algorithms, like Dekker's algorithm [22]). Even though the writes appear before the reads, it is possible for neither thread to observe the other thread's write. Architecturally speaking, one way this could occur is if executing a write places it into a store buffer that is not flushed out to memory until later.

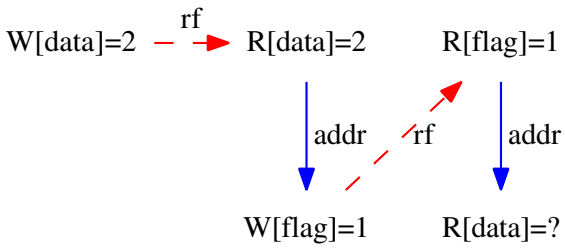
Another case to consider is message passing:

```
                  while (!flag)
data = 1;           continue;
flag = 1;         r = data;
```

Is `r == 0` allowed?

Here, one thread writes a message to the `data` location and then sets `flag` to indicate it is done; the other thread waits until the flag is set and reads the message. On x86 this code works as intended—`r == 0` is not allowed. ARM, however, does allow `r == 0`. Architecturally speaking, one way this could occur is the processor executing instructions out of order. While both x86 and ARM have out-of-order execution, x86 takes pains to avoid getting caught in the act, and ARM does not.

The examples discussed so far can be explained in a fairly straightforward way as the processor executing instructions out of order. This is insufficient to explain all of the behavior allowed by architectures such as POWER. For example, consider this variation on message passing (known as "WRC", for write-to-read causality), which we present diagrammatically as a graph of memory actions:



Here, we write "rf" to indicate a "reads-from" relationship between a write and a read and "addr" to indicate an address dependency. In this example, thread 0 writes to `data`, which is read by thread 1. Thread 1 then sets `flag`, which is read by thread 2, which subsequently reads from `data`. In order to ensure that the instructions are not reordered on by the processor, we insert address dependencies. If all weak memory effects came from the reordering of instructions, thread 2 would need to read 2 from `data`. On ARM and Power machines, however, it is possible for a write to become visible to different processors at different times—in this example, the write to `data` could become visible to thread 1 without becoming visible to thread 2.

This is an important point—relaxed memory can *not* be viewed simply as the result of the reordering of instructions, but must also incorporate a memory subsystem that can lead to weak behavior. One way of looking at it is that we must both account for the order of *execution* of instructions by the processor as well as the order in which writes become *visible* in the memory subsystem.

1.2.2 Compiler problems

Although relaxed memory is commonly blamed on hardware, from a language perspective the compiler is arguably even more at fault. In particular, when compilers break sequential consistency, they generally do it even on single-processor machines (where multiple threads are interleaved based on a timer). A wide variety of compiler transformations that are totally valid in a single threaded setting violate sequential consistency. Worse still, many of these transformations are bread-and-butter optimizations: among them are common subexpression elimination, loop invariant code motion, and dead store elimination. Common subexpression elimination can break sequential consistency in a fairly straightforward way, if it can operate on memory reads:

```
int r1 = *p;          int r1 = *p;
int r2 = *q;    ⟶    int r2 = *q;
int r3 = *p;          int r3 = r1;
```

Here, the second read from `*p` is changed to instead simply reuse the earlier read value, effectively reordering the second and third reads. The way that dead store elimination can break sequential consistency is a bit more subtle:

```
x = 1;
y = 1;    ⟶    y = 1;
x = 2;          x = 2;
```

Here, the compiler sees that the first store to `x` is useless and eliminates it; this can allow another thread to observe the write to `y` without seeing the write to `x` that preceded it.

Loop-invariant code motion (loop hoisting) is particularly troublesome, because hoisting loop-invariant memory accesses can cause basically arbitrary reorderings of loads and stores.

There is another potential difficulty caused by loop hoisting, although it is arguably not a violation of sequential consistency:

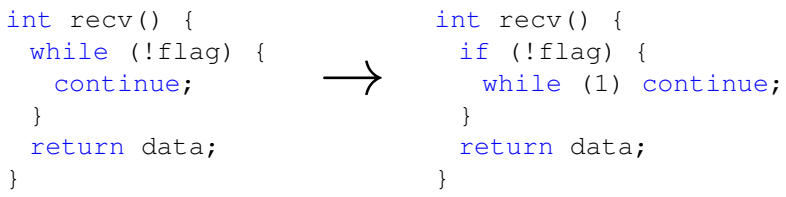
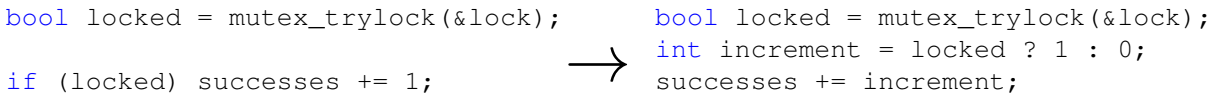


Figure 1.1: Loop hoisting breaking things

By hoisting the load from `flag`, it becomes impossible for `recv` to exit the loop unless the initial read is true. While this is technically consistent with the thread never running again, and thus does not exhibit any behavior inconsistent with the threads interleaving, it is somewhat against the spirit of concurrent computation.

Something that the above examples all have in common is that they can only be detected if multiple threads concurrently access the memory involved, but that is not a necessary condition for an optimization to violate sequential consistency. The primary way to violate sequential consistency for code that is not otherwise racy is to introduce writes into code paths that would not otherwise have them, as in this transformation:



Here, a conditional increment is rewritten into an unconditional one. In the original program, the `successes` variable is only written to if the attempt to lock the mutex succeeded. In the rewritten version, `successes` is always added to, but the value that is added depends on whether the lock succeeded. This introduces a data race (since `successes` is modified even when the lock was not taken); since `+=` is generally implemented by reading the location, adding to it, and then writing it back (and not by an atomic operation), this transformation can allow increments of `successes` to be lost.

Compilers would be interested in this transformation because they can often use a conditional move instruction to select the value to add, avoiding the expense of an actual branch (which may be mispredicted). This optimization was actually performed by some versions of gcc before they walked it back under pressure from the Linux kernel development community [44].

1.3 Language Memory Models

Writing lock-free code in a language without a memory model (like C and C++ until recently) is a fraught affair requiring knowledge of the particular hardware target as well as the specifics of what code the compiler might generate. Indeed, Linux's documentation for kernel programmers devotes significant space to these issues [25, 33]. Even code that scrupulously avoided any unsynchronized accesses could run afoul of of the sort of write-inserting optimizations discussed above. In "Threads Cannot Be Implemented As a Library", Boehm persuasively argues that such an approach is untenable [10].

Language memory models attempt to tame this mess by providing a contract between the language implementer and users. Models provide language users with guarantees about how their code will behave and implementers with clarity about the boundaries of permissible optimization.

The starting point for most language memory models, such as Java's [32] and C++'s [12], is to focus on the common case of programs in which there are no *data races* ("concurrent conflicting accesses"). In this approach, the language requires that mutexes (and other concurrency primitives) be used to to protect accesses to shared data. In exchange, the language promises that programs that do this will be sequentially consistent.

Some decision needs to be made about what semantics to give to programs that *do* still have data races. Generally, the idea is to give weak enough semantics that most traditional optimizations are still valid (as long as they avoid moving memory accesses past synchronization operations in unsafe ways). Typically, the common subexpression elimination, dead store elimination, and loop hoisting transformations shown in 1.2.2 are all valid, as only programs with data races could observe that changes. The branch elimination transformation, however, is not valid, since inserting the unconditional write to `successes` can introduce non-SC behavior in a program without data races.

This is a good starting point for a language memory model. Java and C++ both use it, as does RMC. Given this, there are two major questions left to answer when designing a language memory model: exactly what weak semantics to give programs with data races and what lower-level facilities to provide to allow programming that does not depend on locking everything. The latter question, in particular, will take us deep into the rabbit hole.

1.3.1 Java

Java's answer for what facilities to provide for when mutexes are inappropriate is fairly simple and straightforward: locations declared as "volatile" can be safely accessed concurrently without being considered data races or forfeiting sequential consistency.

Java's story for what to do in programs with data races is substantially less straightforward, however. Java is a safe language that needs to provide some sort of semantics for incorrect (and even for overtly malicious) code. Java has struggled to provide a satisfactory answer here; as published its model prohibits some optimizations that are actually performed and allows some behaviors that were intended to be disallowed [5].

1.3.2 C++11

C++'s answer for what semantics to give programs with data races is simple and very in the spirit of the language: none. Data races are now one of the numerous ways to invoke the specter of "undefined behavior" in C and C++ code.

The facilities provided by C++ for writing lock-free code, on the other hand, are multilayered and quite complicated. The highest level and simplest of these facilities that C++ provides are the *sequentially consistent atomics*—or *SC atomics*— that behave essentially like Java's volatile. The language with just locks and SC atomics admits an extremely clean and simple formalization: we need only consider potential sequentially consistent executions of the program, and if any of those have data races then the behavior is undefined.

Unfortunately, these SC atomics are fairly expensive to implement (requiring a fence for stores even on the extremely programmer friendly x86). Since many algorithms do not require the full guarantees of SC, C++ also provides various forms of *low-level atomics*. The general approach is to allow atomic memory operations to have a "memory order" specified; these memory orders determine what guarantees are provided. In order to model this more permissive system, C++, as is common, treats memory as a set of memory operations related by various partial orders. We will not delve into too much detail, but the most central of these orders is `happens-before`, which serves a key role in determining what writes are visible to reads. The `happens-before` relation, then, is best thought of as the transitive[1] closure of the union of program order and the `synchronizes-with` relation, which models inter-thread communication.

Other than "sequentially consistent", the tamest of these orders are "release" and "acquire". Release and acquire operations are not guaranteed to be sequentially consistent, but do allow threads to synchronize with each other in a fairly straightforward way: when an acquire operation reads from a release operation, a `synchronizes-with` relation holds between the operations, making any writes in the releasing thread visible in the acquiring one. A pair of simple functions to pass a message between two threads can be written using release and acquire:

```
int data;
std::atomic<int> flag;

void send(int msg) {
  data = msg;
  flag.store(1, std::memory_order_release);
}

int recv() {
  while (!flag.load(std::memory_order_acquire))
    continue;
  return data;
}
```

All of the rest gets fairly hairy. The "relaxed" memory order is mostly useful for things like atomic counters and in conjunction with explicit memory fences. The explicit memory fences come in one variety for each of the memory orders (except consume and relaxed) and are intended

[1] Although as we will see shortly, it is not *actually* transitive.

for porting old code and further optimizing the placement of memory barriers. The definition of fences is one of the most complicated bits of the model, and the behavior of the fences is often not as expected. In particular, it is not possible to use "sequentially consistent" fences to restore sequential consistency to a program using weaker memory operations.

The "consume" memory order behaves like "acquire", except that it only establishes a `happens-before` relationship with operations that are data-dependent on the consume operation. This feature is included because on many architectures (such as ARM), an address or data dependency in the receiving thread of a message passing idiom is sufficient to ensure ordering and is often much cheaper than issuing a barrier. Some algorithms critically rely on this property for efficiency. Unfortunately, it is also sort of a mess. First, the simple definition of `happens-before` given above needs to be changed to a much messier form to accommodate this behavior; worse, the new definition isn't transitive! Second, it is dangerous to bake in a syntactic notion of dependency into the semantics of a language. Compiler optimizations work very hard to optimize away unnecessary dependencies, which makes it difficult to preserve consume's guarantees. Consequently, most compilers currently implement consume as equivalent to acquire, sacrificing its potential performance wins. There are proposals by McKenney et al. to change the semantics of consume to be more realistically implementable, but nothing seems finalized yet [35, 36].

C++11's ordering annotations and—especially—fences are coarse-grained tools. Memory order annotations on memory accesses allow the establishment of one-to-many ordering constraints and fences allow the establishment of many-to-many constraints, but there is no way to establish ordering between two individual accesses. If—as we believe is natural—the programmer reasons about the code in terms of relative ordering, this requires the programmer to translate from this mental model to the coarse-grained facilities provided by C++. We believe this is work better left to a compiler.

1.4 A new approach

The core philosophy of the Relaxed Memory Calculus (RMC) approach is that reasoning about the relative ordering of events is fundamental to reasoning about concurrent programming. In a sequentially consistent setting, one might reason about the message passing example as follows "If `flag = 1` is visible, then `data = 1` as well. Furthermore, a read that `flag == 1` executes before `r = data`, so we must have `r == 1`." Since relative ordering of operations is the key concept of low-level concurrent programming, we believe it is natural to directly expose it in the language.

It turns out to be useful to draw some finer distinctions, so in RMC programmers explicitly specify constraints on both the order of visibility of memory writes and on the execution of operations. These constraints are then enforced by the compiler, which has a great deal of latitude in how to achieve them. Because of the very fine-grained information about permissible behaviors, this can allow the generation of more efficient code.

This leads us to our thesis statement: *Explicit programmer-specified constraints on execution order and visibility of writes are a practical approach for low-level concurrent programming in the presence of modern hardware and compiler optimizations.*

For our approach to be practical, it should have an implementation in a real programming language that is rigorous, usable, and efficient. To demonstrate this, we:

- Give a tutorial introduction to RMC-C++, an extension of C++ that provides low-level atomics controlled by explicit programmer-specified ordering constraints (Chapter 2).
- Show that RMC is *rigorous* and amenable to formal reasoning by laying out the formal definition of the RMC core calculus, which has been substantially revised and extended since its original publication (Chapter 3).
- Present `rmc-compiler`, our LLVM-based compiler for RMC-extended languages and discuss the compilation of RMC to x86, ARM, and POWER (Chapter 4).
- Argue that our approach is *usable* by presenting a collection of realistic lock-free data structures and other related low-level concurrent programs implemented in RMC-C++ (Chapter 5).
- Demonstrate our *efficiency* by evaluating the performance of RMC programs (Chapter 6).

Chapter 2

The Relaxed Memory Calculus

In this section, we give a tour of the Relaxed Memory Calculus (RMC), through the lens of its realization as an extension to C++. When we need to distinguish "RMC as realized as a C++ extension" from "RMC as a core calculus", we will sometimes refer to the former as RMC-C++.

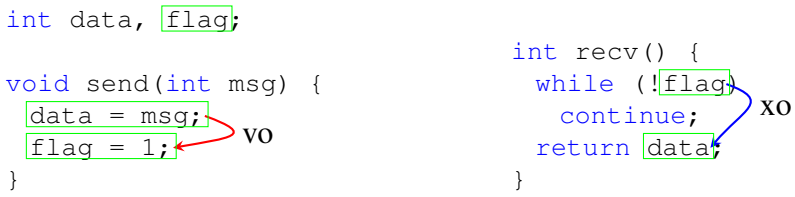
Much of the material in this section is adapted or copied from "A Calculus for Relaxed Memory" by Karl Crary and myself [18].

2.1 A Tour of RMC

2.1.1 Basics

The Relaxed Memory Calculus (RMC) is a different approach to low-level lock-free concurrent programming. In the RMC model, the programmer can explicitly and directly specify the key ordering relations that govern the behavior of the program.

These key relations—which we will also refer to as "edges"—are that of *visibility-order* ($\xrightarrow{\text{vo}}$) and *execution-order* ($\xrightarrow{\text{xo}}$). To see the intended meaning of these relations, consider this pair of simple functions for passing a message between two threads:



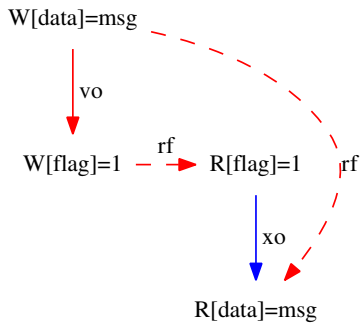
The visibility edge ($\xrightarrow{\text{vo}}$) between the writes in `send` ensures that the write to `data` is visible to other threads before the write to `flag` is. Somewhat more precisely, it means that any thread that can see the write to `flag` can also see the write to `data`. The execution edge ($\xrightarrow{\text{xo}}$) between the reads in `recv` ensures that the reads from `flag` occur before the read from `data` does. This combination of constraints ensures the desired behavior: the loop that reads `flag` can not exit until it sees the write to `flag` in `send`; since the write to `data` must become visible to a thread first, it must be visible to the `recv` thread when it sees the write to `flag`; and then, since the read from `data` must execute after that, the write to `data` must be *visible to* the read.

Architectural aside: Having two actions ordered by visibility order most closely corresponds to requiring that they be separated by a Power-style "lightweight" sync (`lwsync`). A lightweight sync prevents instructions after it from being executed by the processor until all instructions before it have executed as well as preventing any writes after the sync from propagating to some other CPU until all writes that had propagated to the issuing CPU before the sync have propagated to that other CPU.

Having two actions ordered by execution order requires that something be done to prevent the instructions from being executed out of order on the CPU (in an observable way). On some architectures this could include the use of address or control dependencies as well as regular barriers.

On a Total Store Order machine such as the x86, nothing special need be done to enforce these orders. They always hold.

We can demonstrate this this diagrammatically as a graph of memory actions with the constraints as labeled edges:



In the diagram, the programmer specified edges ($\xrightarrow{\text{vo}}$ and $\xrightarrow{\text{xo}}$) are drawn as solid lines while the "reads-from" edges (written $\xrightarrow{\text{rf}}$), which arise dynamically at runtime, are drawn as dashed lines. Since reading from a write is clearly a demonstration that the write is visible to the read, we draw reads-from edges in the same color red as we draw specified visibility-order edges, to emphasize that both carry visibility. Then, the chain of red visibility edges followed by the chain of blue execution order edges means that the write to `data` is *visible to* the read.

2.1.2 Concrete syntax: tagging

Unfortunately, we can't actually just draw arrows between expressions in our source code, and so we need a way to describe these constraints in text. We do this by tagging expressions with names and then declaring constraints between tags:

```
int data;
rmc::atomic<int> flag;

void send(int msg) {
  VEDGE(wdata, wflag);
  L(wdata, data = msg);
  L(wflag, flag = 1);
}
```

```
int recv() {
  XEDGE(rflag, rdata);
  while (!L(rflag, flag))
    continue;
  return L(rdata, data);
}
```

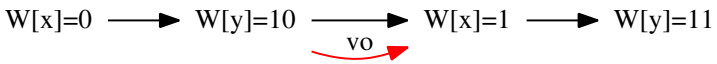
Here, the `L` construct is used to tag expressions. For example, the write `data = msg` is tagged as `wdata` while the read from `flag` is tagged `rflag`. The declaration `VEDGE(wdata, wflag)` creates a visibility-order edge between actions that are tagged `wdata` and actions tagged `wflag`. `XEDGE(rflag, rdata)` similarly creates an execution-order edge.

Visibility order implies execution order, since it does not make sense for an action to be visible before it has occured.

Visibility and execution edges only apply between actions in program order. This is mainly relevant for actions that occur in loops, such as:

```
VEDGE(before, after);
for (i = 0; i < 2; i++) {
  L(after, x = i);
  L(before, y = i + 10);
}
```

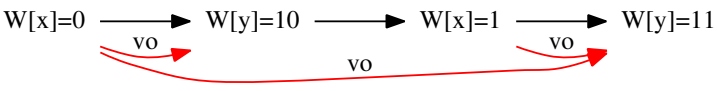
This generates visibility edges from writes to `y` to writes to `x` in future iterations, as shown in this trace (in which unlabeled black lines represent program order):



Furthermore, edge declarations generate constraints between all actions that match the tags, not only the "next" one. If we flip the `before` and `after` tags in the previous example, we get:

```
VEDGE(before, after);
for (i = 0; i < 2; i++) {
  L(before, x = i);
  L(after, y = i + 10);
}
```

which yields the following trace:



In addition to the obvious visibility edges between writes in the same loop iteration, we also have an edge from the write to `x` in the first iteration to the write to `y` in the second. This behavior extends to actions in future invocations of the function, as well. [1] This behavior will be important in the ring buffer example in Section 2.2.1.

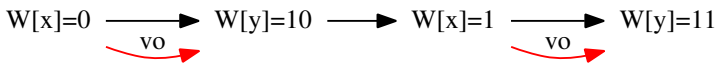
While this behavior is a good default, it is sometimes necessary to have more fine-grained

[1] Though *not* to actions in an overlapping recursive invocation of a function. See Section 2.5.2.

control over which matching actions are constrained. This can be done with "scoped" constraints: `VEDGE_HERE(a, b)` establishes visibility edges between executions of `a` and `b`, but only ones that do not leave the "scope" of the constraint. [2] We can modify the above example with a scoped constraint:

```
for (i = 0; i < 2; i++) {
  VEDGE_HERE(before, after);
  L(before, x = i);
  L(after, y = i + 10);
}
```

which yields the following trace in which the edges between iterations of the loop are not present:



Because of annoying implementation details, `L` may only be used to tag expressions with values that can be assigned to a variable. To tag declarations, blocks, or calls to `void` functions, we provide `LS(name, stmt);`.

2.1.3 Pre and post edges

So far, we have showed how to draw fine-grained constraint edges between actions. Sometimes, however, it is necessary to declare visibility and execution constraints in a much more coarse-grained manner. This is particularly common at library module boundaries, where it would be unwieldy and abstraction breaking to need to specify fine-grained edges between a library and client code. To accommodate these needs, RMC supports special `pre` and `post` labels that allow creating edges between an action and *all* of its program order predecessors or successors.

One of the most straightforward places where coarse-grained constraints are needed are in the implementation of locks. Here, any actions performed during the critical section must be visible to any thread that has observed the unlock at the end of it, as well as not being executed until the lock has actually been obtained. This corresponds to the actual release of a lock being visibility-order *after* everything before it in program order and the acquisition of a lock being execution-order *before* all of its program order successors.

In this example implementation of simple spinlocks, we do this with post-execution edges from the exchange that attempts to acquire the lock and with pre-visibility edges to the write that releases the lock:

[2] Where the "scope" of a constraint is defined (somewhat unusually) as everything that is dominated by the constraint declaration in the control flow graph. Recall that a basic block A dominates B if every path from the entry point to B passes through A

```
void spinlock_lock(spinlock_t *lock) {
  XEDGE(trylock, post);
  while (L(trylock, lock->state.exchange(1)) == 1)
    continue;
}

void spinlock_unlock(spinlock_t *lock) {
  VEDGE(pre, unlock);
  L(unlock, lock->state = 0);
}
```

2.1.4 Transitivity

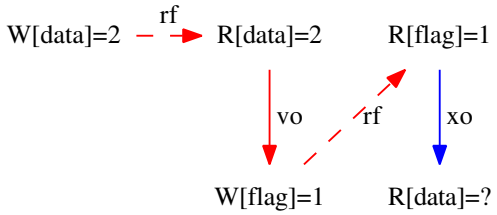
Visibility order and execution order are both transitive. This means that, although the primary meaning of visibility order is in how it relates writes, it is still useful to create edges between other sorts of actions.

In fact, because of this transitivity, it is even sometimes profitable to create edges to no-ops! In the `spinlock_lock` example before, we draw an execution edge from `trylock` to the quasi-tag `post`. This means that each exchange on the lock is execution ordered before not only the body of the critical section, but, if the test-and-set fails, any future test-and-set attempts. This is stronger than is actually necessary, and we can weaken it using a no-op:

```
void spinlock_lock(spinlock_t *lock) {
  XEDGE(trylock, acquired);
  XEDGE(acquired, post);
  while (L(trylock, lock->state.exchange(1)) == 1)
    continue;
  L(acquired, noop());
}
```

Here, the exchange is specified to execute before `acquired`, which is a no-op that is specified to execute before all of its successors. Since a no-op doesn't *do* anything, nothing is needed to ensure the execution order with the no-op, but transitivity ensures that the lock attempts are execution ordered before everything after `acquired`.

A somewhat more substantive and less niche application of transitivity of visibility occurs with visibility edges from reads to writes. Consider the following trace, a variant of the write-to-read causality example discussed earlier:

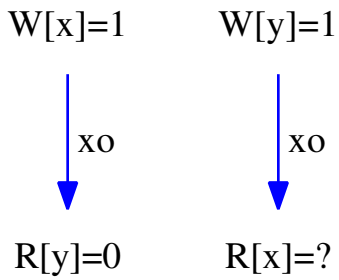


Since reads-from is a form of visibility, and since visibility is transitive, this means that `W[data]=2` is visible before `W[flag]=1`. It is then also visibility ordered before `R[flag]=1`; since

that must execute before `R[data]=?`, this means that `W[data]=2` must be *visible to* `R[data]=?`, which will then read from it.

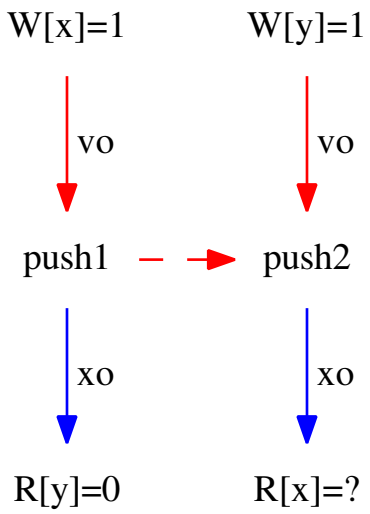
2.1.5 Pushes

Visibility order is a powerful tool for controlling the *relative* visibility of actions, but sometimes it is necessary to worry about *global* visibility. One case where this might be useful is in preventing store buffering behavior:



Here, two threads each write a 1 into a memory location and then attempt to read the value from the other thread's location (this idiom is the core of the classic "Dekker's algorithm" for two thread mutual exclusion). In this trace, `R[y]=0` reads 0, and we would like to require (as would be the case under sequential consistency) that `R[x]=?` will then read 1. However, it too can read 0, since nothing forces `W[x]=1` to be visible to it. Although there is an execution edge from `W[x]=1` to `R[y]=0`, this only requires that `W[x]=1` *executes* first, not that it be visible to other threads. Upgrading the execution edges to visibility edges is similarly unhelpful; a visibility edge from a write to a read is only useful for its transitive effects, and there are none here. What we need is a way to specify that `W[x]=1` becomes *visible* before `R[y]=0` *executes*.

Pushes provide a means to do this: when a push executes, it is immediately globally visible (visible to all threads). As a consequence of this, visibility between push operations forms a total order. Using pushes, we can rewrite the above trace as:



Here, we have inserted a push that is visibility-after the writes and execution-before the read. Since visibility among pushes is total, either push1 or push2 is visible to the other. If push1 is visible before push2, as in the diagram, then `W[x]=1` is visible to `R[x]=?`, which will then read 1. If push2 was visible to push1, then `R[y]=0` would be impossible, as it would be able to see the `W[y]=1` write.

In the concrete syntax, inserting a push takes the form of the simple but cumbersome:

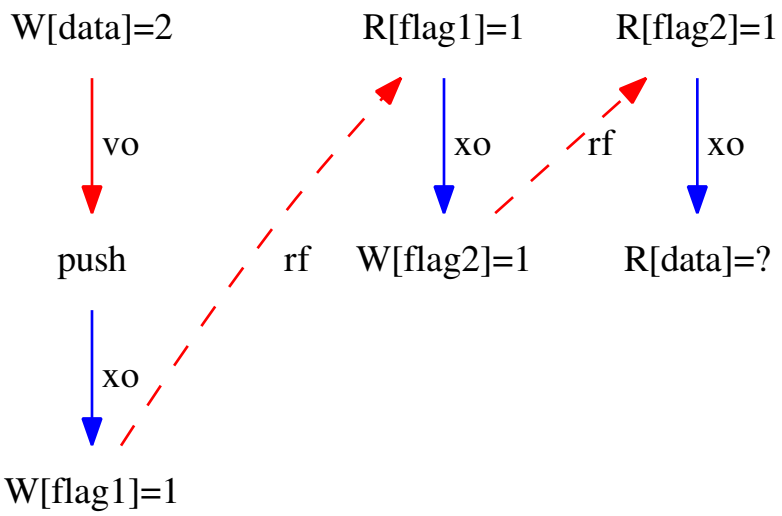
```
VEDGE(write1, push1);
XEDGE(push1, read1);
L(write1, x = 1);
L(push1, rmc::push());
r = L(read1, y);
```

As a convenience, we provide the derived notion of "push edges". A push edge from an action a to b means that a push will be performed that is visibility after a and execution before b. More informally, it means that a will be globally visible before b executes.

```
PEDGE(write1, read1);
L(write1, x = 1);
r = L(read1, y);
```

Architectural aside: Pushes closely match "full" or "heavyweight" fences—`MFENCE` on x86, `sync` on Power, `dmb` on ARM. While in RMC terms, a push is an action that is visible to everything immediately, in an architectural sense, it is probably best thought of as an action that, when executed, stalls until all actions visible-before it have propagated to other threads.

Another, somewhat more unusual example of using pushes is:



Here, `W[data]=2` is visibility before the push. Since the push executes before `W[flag1]=1` and reads can only read-from writes that have already been executed, the push must execute before all of the other actions in this trace save the write to `data`. Since pushes are globally visible to all actions that execute after them, this means that the write to `data` is visible to the read from it, which must then return `2`. If the push was elided and we merely had a visibility edge from `W[data]=2` to `W[flag1]=1`, this would not be the case, and we would require a visibility edge between `R[flag1]=1` and `W[flag2]=1`. Thus, pushes have stronger transitivity (or cumulativity) properties than just plain visibility edges do.

2.2 Example

2.2.1 Ring buffers

As a realistic example of code using the RMC memory model, consider the code in Figure 2.1. This code—adapted from the Linux kernel [26]—implements a ring buffer, a common data structure that implements an imperative queue with a fixed maximum size. The ring buffer maintains front and back pointers into an array, and the current contents of the queue are those that lie between the back and front pointers (wrapping around if necessary). Elements are inserted by advancing the back pointer, and removed by advancing the front pointer.

```
#define BUF_SIZE 1024 /* must divide the range of unsigned */

typedef struct ring_buf {
  unsigned char buf[BUF_SIZE];
  rmc::atomic<unsigned> front;
  rmc::atomic<unsigned> back;
} ring_buf;

bool buf_enqueue(ring_buf *buf, unsigned char c) {
  XEDGE(echeck, insert);
  VEDGE(insert, eupdate);

  unsigned back = buf->back;
  unsigned front = L(echeck, buf->front);

  bool enqueued = false;
  if (back - front < BUF_SIZE) {
    L(insert, buf->buf[back % BUF_SIZE] = c);
    L(eupdate, buf->back = back + 1);
    enqueued = true;
  }
  return enqueued;
}

int buf_dequeue(ring_buf *buf) {
  XEDGE(dcheck, read);
  XEDGE(read, dupdate);

  unsigned front = buf->front;
  unsigned back = L(dcheck, buf->back);

  int c = -1;
  if (back - front > 0) {
    c = L(read, buf->buf[front % BUF_SIZE]);
    L(dupdate, buf->front = front + 1);
  }
  return c;
}
```

Figure 2.1: A ring buffer

This ring buffer implementation is a single-producer, single-consumer, lock-free ring buffer. This means that only one reader and one writer are allowed to access the buffer at a time, but the one reader and the one writer may access the buffer concurrently.

In this implementation, we do not wrap the front and the back indexes around when we increment them, but instead whenever we index into the array. The number of elements in the buffer, then, can be calculated as `back - front`.

There are two important properties we require of the ring buffer: (1) the elements dequeued are the same elements that we enqueued (that is, threads do not read from an array location

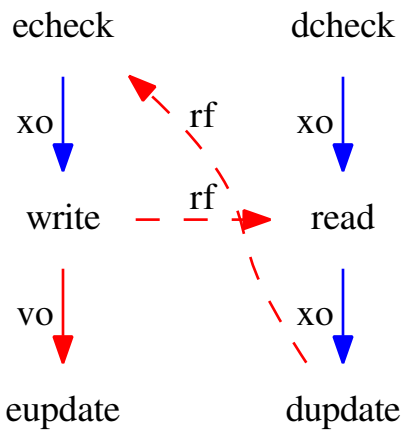


Figure 2.2: Impossible ring buffer trace

without the write to that location being visible to it), and (2) no enqueue overwrites an element that has not been dequeued.

The key lines of code are those tagged `echeck`, `insert`, and `eupdate` (in `enqueue`), and `dcheck`, `read`, and `dupdate` (in `dequeue`). (It is not necessary to use disjoint tag variables in different functions; we do so to make the reasoning more clear.)

For property (1), the key constraints are `insert` $\xrightarrow{\text{vo}}$ `eupdate` and `dcheck` $\xrightarrow{\text{xo}}$ `read`. If we consider an dequeue reading from some enqueue, `dcheck` reads from `eupdate` and so `insert` $\xrightarrow{\text{vo}}$ `eupdate` $\xrightarrow{\text{rf}}$ `dcheck` $\xrightarrow{\text{xo}}$ `read`. Thus `insert` is visible to `read`. Note, however, that if there are more than one element in the buffer, the `eupdate` that `dcheck` reads from will not be the `eupdate` that was performed when this value was enqueued, but one from some *later* enqueue. That is just fine, and the above reasoning still stands. As discussed above, constraints apply to *all* matching actions, even ones that do not occur during the same function invocation. Thus the write of the value into the buffer is visibility ordered before the `back` updates of all future enqueues by that thread.

Property (2) is a bit more complicated. The canonical trace we wish to prevent appears in Figure 2.2. In it, `read` reads from `insert`, a "later" write that finds room in the buffer only because of the space freed up by `dupdate`. Hence, a current entry is overwritten.

This problematic trace is impossible, since `read` $\xrightarrow{\text{xo}}$ `dupdate` $\xrightarrow{\text{rf}}$ `echeck` $\xrightarrow{\text{xo}}$ `insert` $\xrightarrow{\text{rf}}$ `read`. Since you cannot read from a write that has not executed, writes must be executed earlier than any read that reads from them. Thus this implies that `read` executes before itself, which is a contradiction.

2.2.2 Using data dependency

One of the biggest complications of the C++11 model is the "consume" memory order, which establishes ordering based on address and data dependencies (which we will generally lump together as "data dependencies"). This is useful because it allows read access to data structures

to avoid needing any barriers in many cases. This technique is extremely widespread in the Linux kernel. Some example code that could use this technique:

```
// A widget storing library
rmc::atomic<widget *> widgets[NUM_WIDGETS];

void update_widget(char *key, int foo, int bar) {
  VEDGE(init, update);
  widget *w = L(init, new widget(foo, bar));

  int idx = calculate_idx(key);
  L(update, widgets[idx] = w);
}

// Some client code
int use_widget(char *key) {
  XEDGE_HERE(lookup, a);

  int idx = calculate_idx(key);
  widget *w = L(lookup, widgets[idx]);
  return L(a, w->foo) + L(a, w->bar);
}
```

Here, we have a toy library for storing an array of widgets that tries to illustrate the shape of such code. In `update_widget`, a new widget object is constructed and initialized and then a pointer is written into an array; to ensure visibility of the initialization, a visibility edge is used. In `use_widget`, which is a client function to look up a widget and add together its two fields, the message passing idiom is completed by execution edges from the lookup to the actual accesses of the object. The use of `XEDGE_HERE` is the one modification made in order to enable data dependencies—data dependencies can't enforce ordering with *all* subsequent invocations of the function, so we use `XEDGE_HERE` so that the ordering only needs to apply within a given execution of the function. The key thing about this code is that it uses the same execution order idiom as message passing that does not have data dependencies—in RMC, we just provide a uniform execution order mechanism and rely on the compiler to be able to take advantage of existing data dependencies in cases like this one.

(Note that this code totally ignores the issue of freeing old widgets if they are overwritten. This is a subtle issue; the solution generally taken in Linux is the read-copy-update (RCU) mechanism [34].)

2.3 Advanced Features

2.3.1 Non-atomic locations and data races

While atomic memory operations in RMC come with extremely weak semantics (when unconstrained by visibility and execution edges), they still come with requirements that conflict with desired compiler optimizations, and so are too strong to credibly apply to *all* memory accesses. One such requirement is that each read reads the entire value from exactly one write: this can be invalidated when accessing large objects (which often need to be broken into several accesses)

or if updates to multiple objects are merged into a single `memset` or `memcpy` call. Another is the requirement that writes will eventually propagate out to all threads: as demonstrated in Figure 1.1, hoisting a read out of a loop can prevent it from ever observing a write.

While these assumptions are important when writing concurrent algorithms, they are a burden when compiling accesses that can't conflict with others. We thus adopt the same solution for RMC as vanilla C++11 adopts: we introduce a notion of *non-atomic* locations. These non-atomic locations are not allowed to be accessed concurrently, on pain of invoking undefined behavior (halt-and-catch-fire). By "accessed concurrently", we mean that there is a data race: two accesses to a location from different threads form a data race if at least one is a write and neither is visible-to the other. More details about data-races and the rationale for this definition can be found in Section 3.4.6.

In RMC-C++, locations are non-atomic by default. Making a location atomic is done at the type level, by declaring it as an `rmc::atomic<T>`.

Because the language makes *no* guarantees about the behavior of programs with racy accesses to non-atomics, *any* compiler transformation that is only detectable by programs with data-races on non-atomics is permitted: any program that can observe the different behavior induced by the transformation has a data-race on a non-atomic; since its behavior is therefore undefined, whatever behavior it observed is permissible.

2.3.2 Sequentially consistent locations

Rationale

As discussed in Section 1.3.2, the C++11 model for concurrency has three different "levels" of mechanisms, each with increasing complexity and control: mutexes and other high level synchronization objects, sequentially consistent atomics, and low-level atomics.

While most of the focus of RMC—and most of the discussion in this document—has been on enabling low-level programming with fine-grained control a la C++'s low-level atomics, it is important that this be able to coexist with more coarse grained concurrent programming. RMC's story for incorporating locks is compelling and unsurprising—they can be implemented in RMC as shown in 2.1.3 and the compiler can generate efficient code.

Implementing something like C++'s sequentially consistent atomics—in which all operations on them have a total order—is also possible, although with some drawbacks. One implementation is:

```
void sc_store(rmc::atomic<int> *p, int val) {
  PEDGE(pre, write);
  L(write, *p = val);
}

int sc_load(rmc::atomic<int> *p) {
  PEDGE(pre, read);
  XEDGE(read, post);
  return L(read, *p);
}
```

The pushes that occur before the loads and stores ensure that there is a push between any pair of SC operations. As shown in [18], this is sufficient to guarantee sequentially-consistent ordering between these actions. The execution post-edge from the load makes message passing through SC atomics work. The problem with this implementation is that the semantics it gives are *stronger* than necessary. In particular, this implementation requires that all previous memory operations are globally visible before an SC operation executes. This is stronger than is necessary—we need only that all of the other operations that we want to participate in the SC order be globally visible.

This is not merely a theoretical concern: C++'s SC atomics can be implemented on x86 as an `MFENCE` after stores and no fences at all on a load. The stronger RMC-style SC atomics presented above, however, require emitting an `MFENCE` before both loads and stores, which can be a substantial performance penalty. There are some other approaches for implementing SC atomics on top of already discussed RMC facilities, but all have similar problems.

Design

Once we have conceded that we actually need some kind of built in sequentially consistent operation, the high-level design is straightforward. At the language level, we simply introduce a new type of location, written `rmc::sc_atomic<T>`, such that all operations on SC atomics participate in a total order of SC operations that is consistent with program order and is respected by reads (that is, reads read from the most recent write in the order). [3] Details about how this is achieved formally are discussed in Section 3.4.5. Additionally, so that sequentially consistent operations have the appropriate message passing semantics when interacting with weaker operations, all SC stores are implicitly visibility-after all earlier actions and SC loads are implicitly execution-before all later actions.

2.3.3 Give and take - fine-grained cross function edges

Motivation

So far, we lack any kind of fine-grained support for constraint edges between different functions. Currently, ordinary point-to-point constraint edges can only be specified between two actions that appear inside the same function body. Thus, we are left with two rather indirect ways to impose edges between actions in different functions: place a label on an entire function invocation or use `pre`/`post` constraints. These are both very coarse-grained, however, and constrain *all* of the memory operations within the other function. This is fine for many situations, since most techniques for ensuring ordering are many-to-many or one-to-many, but some algorithms depend heavily on taking advantage of the point-to-point ordering guarantees provided by data dependence.

[3] Another approach, which is actually the one taken in the core calculus, is to instead allow each individual operation to be tagged as either SC or not. We don't adopt this for our C++ version of RMC for several reasons—it complicates the programming model, it is probably generally not good practice to mix these access types anyways, and there have been correctness issues in C++11 regarding this sort of mixing that we would sooner avoid.

One example of this is the code presented in 2.2.2. That example is designed to demonstrate taking advantage of data dependencies, but the code is actually structured fairly poorly: `use_widget` contains both the code for fetching the widget from the data structure and the code that uses the widget. This is unfortunate—we would rather write something like:

```
// Some library
widget *get_widget(char *key) {
  int idx = calculate_idx(key);
  return L(get_widget_lookup, widgets[idx]);
}

// Some client code
int use_widget(char *key) {
  XEDGE(get_widget_lookup, a);
  widget *w = get_widget(key);
  return L(a, w->foo) + L(a, w->bar);
}
```

Here, the widget lookup is moved into a library function and a cross-function edge is drawn from the lookup to the uses of the widget. We do not, however, want to actually allow the direct specification of cross function edges: without them, RMC compilation is a per-function affair, which is a huge boon for efficient implementation.

Approach

Instead, we adopt the approach of considering the passing of an argument to a function or the returning of a value as a sort of pseudo-action that can be labeled and have edges drawn to it. We call these pseudo-actions "transfers". We write `LGIVE(a, expr)` to give a name to the passing of a value to another function and `LTAKE(a, expr)` to name the receipt of a value from another function. The above code would then be written as:

```
// Some library
widget *get_widget(char *key) {
    XEDGE_HERE(get, ret);
    int idx = calculate_idx(key);
    widget *w = L(get, widgets[idx]);
    return LGIVE(ret, w);
}

// Some client code
int use_widget(char *key) {
    XEDGE_HERE(load_widget, a);

    widget *w = LTAKE(load_widget, get_widget(key));
    return L(a, w->foo) + L(a, w->bar);
}
```

Here, `get_widget` looks up a widget by its id and returns it, but it uses `LGIVE` to specify that the loading of the widget ought to be execution order before returning it. Then, on the `use_widget` side, we use `LTAKE` to indicate that reading out of the widget must execute after the transfer that returns the widget from `get_widget`. By the transitivity of execution order, then, this gives us

that the load from `widgets[idx]` in `get_widgets` must be execution-before the reads through `w` in `use_widget` that consume its value.

It is important to note that `LGIVE(a, expr)` assigns a label to the act of passing the value of `expr` to another function (via a parameter or return value). It does *not* label the evaluation of `expr`. Thus it is necessary to separately label the actions producing a return value and the actual return and to link them up with constraints as needed.

We can also invert things, and draw an edge to the action of passing an argument to a function:

```
int consume_widget(widget *w) {
   LTAKE(load_widget, w);
   XEDGE_HERE(load_widget, ret);
   return L(ret, w->foo);
}

int pass_widget() {
   XEDGE_HERE(get, pass);
   widget *w = L(get, widgets[0]);
   return consume_widget(LGIVE(pass, w));
}
```

Here, note that we give a name to the receipt of a parameter by doing an `LTAKE` on the parameter name itself.

Rationale

One perhaps unusual thing is that while the focus of our examples was functions where the argument being passed is closely related to the actions that it is linked to, there is nothing fundamental that requires this. It is entirely possible to use a transfer to draw an edge between totally unrelated actions. If the mechanism is this general, why have the limitation of requiring that these pseudo-actions be connected to argument passing and value returning at all?

This design was chosen because it is a good match for the expected actual use cases of fine-grained cross-function edges: maintaining execution order between reading a pointer and reading through that pointer. In those cases, the edge really is closely connected to a value, and so directly associating it provides a very useful hint for both the programmer and the compiler.

2.3.4 `LPRE` and `LPOST` - Pre and Post edges to other actions

In Section 2.1.4, we demonstrated combining transitivity and no-op actions in order to create a constraint from an action to everything that is program-order after some *other* point in the code. Explicitly using no-op actions is somewhat kludgy and syntactically heavyweight, so we provide `LPRE` and `LPOST` as conveniences around them. For example:

```
void spinlock_lock(spinlock_t *lock) {
  XEDGE(trylock, critical_section);
  while (L(trylock, lock->state.exchange(1)) == 1)
    continue;
  LPOST(critical_section);
}
```

`LPRE` and `LPOST` themselves are just simple macros that label a no-op action and give it either a pre- or post- visibility constraint:

```
#define LPRE(label) do { VEDGE_HERE(pre, label); L(label, noop()); } while(0)
#define LPOST(label) do { VEDGE_HERE(label, post); L(label, noop()); } while(0)
```

We always use visibility edges because visibility implies execution, which in the example above gives us transitive execution edges between `trylock` and everything after the `LPOST`. But edges to no-ops have no meaning on their own, so there are no real visibility constraints.

2.4 Model Details

The model of how an RMC program is executed is split into two parts. On one side, the *execution model* models the potentially out-of-order execution of actions in the program. On the other side, the *memory system model* determines what values are read by memory reads.

Parts of the division of labor between the execution and memory system side in RMC are somewhat unusual for an operational model: the execution model is *extremely* weak and relies on the memory subsystem's coherence rules to enforce the correctness of single-threaded code.

2.4.1 Execution Model

Intuitively, the model that we would like to have for how actions are executed, is that actions may be executed in any order at all, except as constrained by execution and visibility constraints. This includes when the actions have a dependency between them, whether control or data! That is, actions (both reads and writes) may be executed speculatively under the assumption that some "earlier" read will return a particular value.

Unfortunately, this extremely permissive model is *too* weak, and gives rise to problematic and bizarre behaviors known as "thin-air reads" in which writing a speculated value is eventually used to justify the read that produced it, in a sort of bizarre "stable time loop." [4] Unfortunately, while there exist recent systems that seem to solve this problem [29], they have not been incorporated into the RMC formalism yet. The situation is discussed in further detail in Section 3.2.4.

We are left for now, then, with the handwavy prose rule in from the C++14 standard: "Implementations should ensure that no 'out-of-thin-air' values are computed that circularly depend on their own computation." [28].

2.4.2 Memory system model

The main question to be answered by a memory model is: for a given read, what writes is it permitted to read from? Under sequential consistency, the answer is "the most recent", but this is not a necessarily a meaningful concept in relaxed memory settings.

While RMC does not have a useful *global* total ordering of actions, there does exist a useful *per-location* ordering of writes that is respected by reads. [5] This order is called *coherence*

[4] Like the song Johnny B. Goode in "Back to the Future"

[5] **Architectural aside:** This is a direct consequence of cache coherence at the architectural level.

order. It is a strict partial order that only relates writes to the same location, and it is the primary technical device that we use to model what writes can be read. A read, then, can read from any write executed before it such that the constraints on coherence order are obeyed.

As part of RMC's aims to be as weak as possible (but not weaker), the rules for coherence order are only what are necessary to accomplish three goals: First, each individual location's should have a total order of operations that is consistent with program order (this is *slightly* stronger than just what is required to make single-threaded programs work.) Second, message passing using visibility and execution order constraints should work. Third, read-modify-write operations (like test-and-set and fetch-and-add) should be appropriately atomic.
The constraints on coherence order are the following:

- A read must read from the most recent write that it has seen—or from some other coherence-later write. More precisely, if a read A reads from some write B, then any other write to that location that is *prior to* A must be coherence-before B.
- If a write A is *prior to* some other write B to the same location, A must be coherence-before B.
- If a read-modify-write operation A reads from a write B, then A must *immediately* follow B in the coherence order. That is, any other write that is coherence-after A must also be coherence-after B.

An action A is *prior to* some action B on the same location if any of the following holds:

- A is earlier in program order than B. (This is crucial for making single threaded programs work properly.)
- A is *visible to* B.
- A and B are both sequentially-consistent operations and A was executed before B.
- A is transitively ordered before B by the previous three rules.

We've discussed *visible to* already, but more precisely, A is *visible to* B if:

- There is some action X such that A is transitively visibility-ordered before X and X is transitively and reflexively execution-ordered before B. That is, there is a chain of one or more visibility edges followed by a chain of zero or more execution edges from A to B (as in the message passing diagrams). Recall that reads-from and pushes both induce visibility edges.

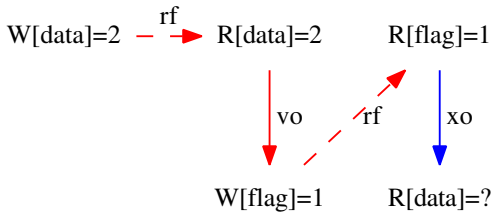
2.5 Discussion

2.5.1 Execution vs. visibility

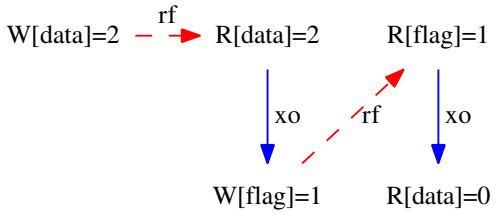
One of the more subtle points of RMC is understanding what precisely the difference between visibility and execution order is.

Visibility order is concerned with controlling the order in which writes become visible in other threads while execution is concerned with, once a write has become visible to one action in a thread, what other actions in *that thread* it is visible to. Visibility constraints imply execution ones.

Visibility, then, has a sort of inter-thread transitivity that execution order does not have. Recall the "WRC" example discussed in Section 2.1.4:



Here, the visibility ordering in the second thread causes `W[data]=2` to be *visible to* `R[data]=?`, which will have to read from it. If we weaken that constraint to execution order, the *visible to* relation is broken, and `R[data]=?` needn't read from `W[data]=2`:



This version corresponds to the variant of WRC discussed in Section 1.2.1 (which used address dependencies—which ensure execution order though not visibility order). While not extending visibility to other threads, execution edges from reads to writes still do have important meaning. As shown in the ring buffer example in Section 2.2.1, this sort of execution constraint can be used to rule out undesired behaviors by causing them to need a cycle in the order actions were executed. Additionally, such constraints play an important role in important role in establishing coherence order. If some other write to `flag` was visible to `R[data]=2`, it would precede `W[flag]=1` in coherence order.

As it turns out, this situation—constraints from a read to a write—is the main place where there is an interesting distinction between visibility and execution. [6] As will be discussed in more detail in Section 4.4.2, other than their intra-thread transitive effects, execution edges from writes have no effect and visibility edges to reads have no more force than execution edges.

This means that we could remove execution edges from the language (using visibility edges in their place) and the only impact would be the inability to specify constraints from reads to writes as execution ordering rather than the stronger visibility. This would come as a nontrivial performance cost on many architectures, though, and we believe the distinction is a useful and natural one. As discussed in Section 1.2.1, however, relaxed memory behavior must be seen as arising from both instruction reordering and a relaxed memory subsystem—this is directly reflected in the split between execution and visibility constraints.

2.5.2 Recursion

One wart of RMC-C++ is that while constraints apply to actions in subsequent invocations of the function, they do *not* apply to recursive invocations of the function. It is probably best to avoid

[6] The other distinction being that write-to-write execution edges have no effect.

writing recursive RMC-C++ functions that use non-scoped constraints. Fortunately, it seems rare that recursion is the natural solution for the sorts of problems that require low-level concurrency, so this entire issue causes little trouble.

This is an ugly decision made for ugly implementation reasons: supporting constraints in recursive functions means that any function call that could possibly call back into the original function (which, absent some serious analyses—and perhaps even with them—will be "most of them") must be treated as a potential branch to the start of the function as well as a potential branch target from the end of the function. This would lead to the insertion of spurious memory barriers in a large number of cases.

Since the downsides of not supporting constraints to recursive invocations of functions seem very minor in practice, we decided it wasn't worth the cost or the complexity of serious efforts to mitigate the cost.

Chapter 3

RMC Formalism

RMC is formalized as a core calculus that is used to model possible executions. The dynamic semantics of the calculus explicitly model out-of-order execution and speculation, and do so in a way that attempts to be as weak as possible, to account for future innovations in the design of hardware.

The metatheory for the RMC core calculus is formalized in Coq and includes proofs of type safety as well as proofs that sequential consistency can be recovered by appropriate programming disciplines: either by ruling out data races or by inserting pushes in between all memory accesses.

3.1 Basic RMC

In this section, we present a simple initial version of the RMC formalism that closely corresponds to the original published version [18], but with some improvements (from RMC 2.0 [19]) that greatly simplify the machinery. This version of RMC, which we will call "Basic RMC", lacks several important features and suffers from the Thin-Air Problem.

3.1.1 Syntax

The syntax of RMC is given in Figure 3.1. Tags (T) range over actual dynamically generated tags; they do not appear in user programs. There are two sorts of variables: x ranges over values and t ranges over tags.

We make a standard monadic syntactic distinction between pure terms (m) and effectful expressions (e). The terms language is totally standard; here it is presented with unit types, natural numbers, and recursive functions, but it could be extended with any standard functional features with no complication.

The monadic framework is fairly standard. Expressions are sequenced using $x \leftarrow e_1 \,\mathsf{in}\, e_2$, which executes e_1, binds its value to x, and then executes e_2. Pure terms can be converted into an expression with $\mathsf{ret}\, m$. Effectful computations may be suspended inside pure terms with $\mathsf{susp}\, e$, yielding terms of type $\tau\,\mathsf{susp}$ and a suspended computation can be executed as an expression with $\mathsf{force}\, m$.

The expression forms R, W, and RMW perform memory reads, writes, and read-modify-writes, respectively. Pushes and nops, unsurprisingly, are performed by Push and Nop. The expression RMWS is a technical device for the implementation of RMW; it does not appear in user-written programs and will be discussed later.

The tag allocation form new $t \overset{b}{\Rightarrow} t'.e$ generates two new tags bound to variables t and t' in the scope e. Those tags express either a visibility or execution edge, as indicated by the attribute b.[1] The labeling form $\varphi \# e$ attaches a *label* to an expression, which is either a tag or a quasi-tag $\overset{b}{\triangleleft}$ and $\overset{b}{\triangleright}$ (representing `pre` and `post`).

Finally, an execution state is an association list, pairing thread identifiers (p) with the current expression on that thread.

types	τ	::=	$\mathsf{unit} \mid \mathsf{nat} \mid \tau\ \mathsf{ref} \mid \tau\ \mathsf{susp} \mid \tau \to \tau$
numbers	n	::=	$0 \mid 1 \mid \cdots$
tags	T	::=	$\cdots$
locations	ℓ	::=	$\cdots$
threads	p	::=	$\cdots$
terms	m	::=	$x \mid \texttt{()} \mid \ell \mid n \mid \mathsf{ifz}\ m\ \mathsf{then}\ m\ \mathsf{else}\ m$
		$\mid$	$\mathsf{susp}\, e \mid \mathsf{fun}\ x(x{:}\tau){:}\tau.m \mid m\, m$
values	v	::=	$x \mid \texttt{()} \mid \ell \mid n \mid \mathsf{susp}\, e \mid \mathsf{fun}\ x(x{:}\tau){:}\tau.m$
attributes	b	::=	$\mathsf{vis} \mid \mathsf{exe} \mid$
labels	φ	::=	$t \mid T \mid \overset{b}{\triangleleft} \mid \overset{b}{\triangleright}$
expr's	e	::=	$\mathsf{ret}\, m \mid x \leftarrow e\ \mathsf{in}\ e$
		$\mid$	$\mathsf{force}\, m$
		$\mid$	$\mathsf{R}[m] \mid \mathsf{W}[m, m] \mid \mathsf{RMW}[m, x{:}\tau.m]$
		$\mid$	$\mathsf{RMWS}[\ell, v, m]$
		$\mid$	$\mathsf{Push} \mid \mathsf{Nop}$
		$\mid$	$\mathsf{new}\ t \overset{b}{\Rightarrow} t.e \mid \varphi \# e$
execution states	ξ	::=	$\epsilon \mid \xi \parallel p : e$
tag sig	Υ	::=	$\epsilon \mid \Upsilon, T$
loc'n sig	Φ	::=	$\epsilon \mid \Phi, \ell{:}\tau$
contexts	Γ	::=	$\epsilon \mid \Gamma, t{:}\mathsf{tag} \mid \Gamma, x{:}\tau$

Figure 3.1: RMC Syntax

[1]Thus, in the RMC formalism, each tag has exactly one partner. The more flexible mechanism allowed in the concrete syntax can be interpreted as, for each edge declaration, applying the appropriate tag to each expression labeled with the matching name.

3.1.2 Thread static semantics

The static semantics of RMC expressions and terms is largely standard. They are typechecked relative to two ambient signatures and a context. The ambient signatures specify the valid tags and locations, giving the types for locations. Labels are checked by validating that tag variables appear in the context and that literal tags appear in the tag signature. Execution states are checked by checking each of the constituent expressions. They are not passed a context, because well-formed execution states must be closed.

$\boxed{\Upsilon; \Phi; \Gamma \vdash m : \tau}$

$$\frac{\Gamma(x) = \tau}{\Upsilon; \Phi; \Gamma \vdash x : \tau} \qquad \frac{}{\Upsilon; \Phi; \Gamma \vdash \texttt{()} : \mathsf{unit}} \qquad \frac{\Phi(\ell) = \tau}{\Upsilon; \Phi; \Gamma \vdash \ell : \tau\ \mathsf{ref}}$$

$$\frac{\Upsilon; \Phi;\ \Gamma \vdash e : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{susp}\, e : \tau\ \mathsf{susp}} \qquad \frac{}{\Upsilon; \Phi; \Gamma \vdash n : \mathsf{nat}}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \mathsf{nat} \quad \Upsilon; \Phi; \Gamma \vdash m_2 : \tau \quad \Upsilon; \Phi; \Gamma \vdash m_3 : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{ifz}\ m_1\ \mathsf{then}\ m_2\ \mathsf{else}\ m_3 : \tau}$$

$$\frac{\Upsilon; \Phi; (\Gamma, f{:}(\tau_1 \to \tau_2), x{:}\tau_1) \vdash m : \tau_2 \quad f, x \notin \mathsf{Dom}(\Gamma)}{\Upsilon; \Phi; \Gamma \vdash \mathsf{fun}\ f(x{:}\tau_1){:}\tau_2.m : \tau_1 \to \tau_2}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \tau \to \tau' \quad \Upsilon; \Phi; \Gamma \vdash m_2 : \tau}{\Upsilon; \Phi; \Gamma \vdash m_1 m_2 : \tau'}$$

$\boxed{\Upsilon; \Gamma \vdash \varphi : \mathsf{label}}$

$$\frac{t{:}\mathsf{tag} \in \Gamma}{\Upsilon; \Gamma \vdash t : \mathsf{label}} \qquad \frac{T \in \Upsilon}{\Upsilon; \Gamma \vdash T : \mathsf{label}}$$

$$\frac{}{\Upsilon; \Gamma \vdash \overset{b}{\triangleleft} : \mathsf{label}} \qquad \frac{}{\Upsilon; \Gamma \vdash \overset{b}{\triangleright} : \mathsf{label}}$$

$\boxed{\Upsilon; \Phi; \Gamma \vdash e : \tau}$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{ret}\, m : \tau} \qquad \frac{\Upsilon; \Phi; \Gamma \vdash m : \tau\ \mathsf{susp}}{\Upsilon; \Phi; \Gamma \vdash \mathsf{force}\, m : \tau}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash e_1 : \tau_1 \quad \Upsilon; \Phi; (\Gamma, x{:}\tau_1) \vdash e_2 : \tau_2 \quad x \notin \mathsf{Dom}(\Gamma)}{\Upsilon; \Phi; \Gamma \vdash x \leftarrow e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m : \tau\ \mathsf{ref}}{\Upsilon; \Phi; \Gamma \vdash \mathsf{R}[m] : \tau} \qquad \frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \tau\ \mathsf{ref} \quad \Upsilon; \Phi; \Gamma \vdash m_2 : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{W}[m_1, m_2] : \mathsf{unit}}$$

$$\frac{\Upsilon;\Phi;\Gamma \vdash m_1 : \tau\ \mathsf{ref} \quad \Upsilon;\Phi;\Gamma, x{:}\tau \vdash m_2 : \tau}{\Upsilon;\Phi;\Gamma \vdash \mathsf{RMW}[m_1, x{:}\tau.m_2] : \tau} \qquad \frac{\Upsilon;\Phi;\Gamma \vdash \ell : \tau\ \mathsf{ref} \quad \Upsilon;\Phi;\vdash v : \tau \quad \Upsilon;\Phi;\vdash m : \tau}{\Upsilon;\Phi;\Gamma \vdash \mathsf{RMWS}[\ell, v, m] : \tau}$$

$$\frac{}{\Upsilon;\Phi;\Gamma \vdash \mathsf{Push} : \mathsf{unit}} \qquad \frac{}{\Upsilon;\Phi;\Gamma \vdash \mathsf{Nop} : \mathsf{unit}}$$

$$\frac{\Upsilon;\Phi;(\Gamma, t{:}\mathsf{tag}, t'{:}\mathsf{tag}) \vdash e : \tau \quad t, t' \notin \mathsf{Dom}(\Gamma)}{\Upsilon;\Phi;\Gamma \vdash \mathsf{new}\ t \overset{b}{\Rightarrow} t'.e : \tau} \qquad \frac{\Upsilon;\Gamma \vdash \varphi : \mathsf{label} \quad \Upsilon;\Phi;\Gamma \vdash e : \tau}{\Upsilon;\Phi;\Gamma \vdash \varphi \# e : \tau}$$

$\boxed{\Upsilon;\Phi;\vdash \xi\ \mathsf{ok}}$

$$\frac{}{\Upsilon;\Phi \vdash \epsilon\ \mathsf{ok}} \qquad \frac{p \notin \xi \quad \Upsilon;\Phi \vdash \xi\ \mathsf{ok} \quad \Upsilon;\Phi;\epsilon \vdash e : \tau}{\Upsilon;\Phi \vdash \xi \parallel p : e\ \mathsf{ok}}$$

3.1.3 Thread dynamic semantics

Threads and the store communicate by synchronously agreeing on a *transaction*. The empty transaction ($\emptyset$)—which most thread transitions use—indicates there is no store effect. The initiation transaction $\varphi_1, \ldots, \varphi_n \# i = \alpha \downarrow v$ indicates that the action α is being initiated with labels $\varphi_1, \ldots, \varphi_n$, is assigned identifier i, and is speculatively returning the value v. The edge transaction $T \overset{b}{\Rightarrow} T'$ indicates that T and T' are fresh tags, expressing a b edge.

$$\begin{array}{llll} \text{identifiers} & i & ::= & \cdots \\ \text{actions} & \alpha & ::= & \mathsf{R}[\ell] \mid \mathsf{W}[\ell, v] \mid \mathsf{RW}[\ell, v] \\ & & \mid & \mathsf{Push} \mid \mathsf{Nop} \\ \text{transactions} & \delta & ::= & \emptyset \mid \vec{\varphi} \# i = \alpha \downarrow v \mid T \overset{b}{\Rightarrow} T \mid m : \tau \end{array}$$

$\boxed{e \overset{\delta}{\mapsto} e'}$

The heart of the system is the execution of actions and the handling of tags. When a memory action has had its subterms fully evaluated (so that it is a syntactic action as shown above), it is initiated, synchronizing on the transaction $\epsilon \# i = \alpha \downarrow v$, and is replaced by the speculated value $\mathsf{ret}\, v$. When an initiation transaction is executed, it collects all of the enclosing labels as part of the transaction. Once an expression has been evaluated to a return, the labels can be removed. New constraints synchronize on a transaction that introduces the new tags T and T', and the tags are substituted in for the tag variables.

$$\varphi \# \delta = \begin{cases} (\varphi, \vec{\varphi}) \# i = \alpha \downarrow v & \text{if } \delta = (\vec{\varphi} \# i = \alpha =\downarrow v) \\ \delta & \text{otherwise} \end{cases}$$

$$\frac{}{\alpha \overset{\epsilon \# i = \alpha \downarrow v}{\longmapsto} \mathsf{ret}\, v} \qquad \frac{e \overset{\delta}{\mapsto} e'}{\varphi \# e \overset{\varphi \# \delta}{\longmapsto} \varphi \# e'} \qquad \frac{}{\varphi \# \mathsf{ret}\, v \overset{\emptyset}{\mapsto} \mathsf{ret}\, v}$$

$$\frac{}{\mathsf{new}\ t \overset{b}{\Rightarrow} t'.e \overset{T \overset{b}{\Rightarrow} T'}{\longmapsto} [T, T'/t, t']e}$$

Evaluation of monadic sequencing propagates transactions up when the left hand side is evaluated. Otherwise most of the rest of the rules are fairly straightforward.

$$\frac{e_1 \overset{\delta}{\mapsto} e_1'}{(x \leftarrow e_1 \ \mathsf{in}\ e_2) \overset{\delta}{\mapsto} (x \leftarrow e_1' \ \mathsf{in}\ e_2)} \qquad \frac{}{(x \leftarrow \mathsf{ret}\, v \ \mathsf{in}\ e) \overset{\emptyset}{\mapsto} [v/x]e}$$

$$\frac{m \mapsto m'}{\mathsf{ret}\, m \overset{\emptyset}{\mapsto} \mathsf{ret}\, m'}$$

$$\frac{m \mapsto m'}{\mathsf{force}\, m \overset{\emptyset}{\mapsto} \mathsf{force}\, m'} \qquad \frac{}{\mathsf{force}(\mathsf{susp}\, e) \overset{\emptyset}{\mapsto} e}$$

$$\frac{m \mapsto m'}{\mathsf{R}[m] \overset{\emptyset}{\mapsto} \mathsf{R}[m']}$$

$$\frac{m_1 \mapsto m_1'}{\mathsf{W}[m_1, m_2] \overset{\emptyset}{\mapsto} \mathsf{W}[m_1', m_2]} \qquad \frac{m_2 \mapsto m_2'}{\mathsf{W}[v_1, m_2] \overset{\emptyset}{\mapsto} \mathsf{W}[v_1, m_2']}$$

Read-modify-write instructions get a little tricky. Conceptually, the expression $\mathsf{RMW}[m_l, x{:}\tau.m]$ reads from the location m_l, substitutes the value for x in m, evaluates m, and then writes back the resulting value to m_l in one atomic operation. To implement this, we speculatively guess the value at m_l, compute the new value, and then initiate a read-write (RW) action to perform the actual memory operation. When doing this, it is necessary to ensure that the value we speculated actually matches the value returned from the read-write; to facilitate this we use RMWS expressions, which record the speculated value. To evaluate a $\mathsf{RMW}[\ell, x{:}\tau.m]$ expression, then, we speculatively choose some value v of type τ, and step to an RMWS expression that records v and $[v/x]m$. Once the value to write back is computed, we initiate a read-write action that writes the new value and is speculated to read the speculated old value.

When doing the original value speculation, it is important for type-safety that the value v we make up *actually* has type τ. To ensure this, it uses a typing transaction: $v{:}\tau$. For the transaction to be matched up with the store, v must be a closed term of type τ under the current tag and location signatures. It would also be possible to design the system to do this directly by having the thread dynamic semantics judgment include typing signatures, and directly appealing to the statics. Pushing it out to a transaction allows us to avoid that dependency.

The semantics of read-modify-writes presented here is extremely general, supporting arbitrary pure computations to determine the value to write. In fact, the computation may even diverge without sacrificing any metatheoretic properties. In the actual implementation, however, the set of available read-modify-write operations is limited to a small subset of straightforward arithmetic and bitwise operations and compare-and-swap (though compare-and-swap needs some improvements to the formalism in order to be properly modeled—see Section 3.4.1).

$$\frac{}{\mathsf{RMW}[\ell, x{:}\tau.m] \overset{v:\tau}{\longmapsto} \mathsf{RMWS}[\ell, v, [v/x]m]} \qquad \frac{}{\mathsf{RMWS}[\ell, v_s, v_m] \overset{\epsilon \# i = \mathsf{RW}[l, v_m] \downarrow v_s}{\longmapsto} \mathsf{ret}\, v_s}$$

$$\frac{m_1 \mapsto m_1'}{\mathsf{RMW}[m_1, x{:}\tau.m_2] \overset{\emptyset}{\mapsto} \mathsf{RMW}[m_1', x{:}\tau.m_2]} \qquad \frac{m \mapsto m'}{\mathsf{RMWS}[\ell, v, m] \overset{\emptyset}{\mapsto} \mathsf{RMWS}[\ell, v, m']}$$

$\boxed{\xi \overset{\delta@p}{\longmapsto} \xi'}$

To step an execution state, we can step any one of the individual threads.

$$\frac{e \overset{\delta}{\mapsto} e'}{(\xi \parallel p : e) \overset{\delta@p}{\longmapsto} (\xi \parallel p : e')} \qquad \frac{\xi \overset{\delta@p'}{\longmapsto} \xi'}{(\xi \parallel p : e) \overset{\delta@p'}{\longmapsto} (\xi' \parallel p : e)}$$

The dynamic semantics of terms is completely standard call-by-value evaluation.

$\boxed{m \mapsto m'}$

$$\frac{m_1 \mapsto m_1'}{\mathsf{ifz}\, m_1 \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3 \mapsto \mathsf{ifz}\, m_1' \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3}$$

$$\frac{}{\mathsf{ifz}\, 0 \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3 \mapsto m_2} \qquad \frac{}{\mathsf{ifz}\, s(n) \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3 \mapsto m_3}$$

$$\frac{m_1 \mapsto m_1'}{m_1 m_2 \mapsto m_1' m_2} \qquad \frac{m_{\scriptscriptstyle 1} \to m_2'}{v_1 m_2 \mapsto v_1 m_2'}$$

$$\frac{}{(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.m) v \mapsto [(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.m), v/f, x]m}$$

3.1.4 The Store

RMC's store is modeled in spirit [2] after the storage subsystem of Sarkar *et al.*'s [40] model for Power, with adaptations made for RMC's generality. As in Sarkar *et al.*, the store is represented, not by a mapping of locations to values, but by a *history* of all the events that have taken place. The syntax of events and histories is:

$$\begin{array}{llll} \text{events} & \theta & ::= & \mathsf{init}(i,p) \mid \mathsf{is}(i,\alpha) \mid \mathsf{spec}(i,v) \\ & & \mid & \mathsf{label}(\varphi,i) \mid \mathsf{edge}(b,T,T) \mid \mathsf{exec}(i) \mid \mathsf{rf}(i,i) \\ & & \mid & \mathsf{co}(i,i) \\ \text{histories} & H & ::= & \epsilon \mid H,\theta \end{array}$$

[2] "*Inspired* by a true story"

Four events pertain to the initiation of actions: $\mathsf{init}(i, p)$ records that i was initiated on thread p, $\mathsf{is}(i, \alpha)$ records that i represents action α, $\mathsf{spec}(i, v)$ records that i was speculated to return the value v, and $\mathsf{label}(\varphi, i)$ records that label φ is attached to i. These events always occur together, but it is technically convenient to treat them as distinct events.

The event $\mathsf{edge}(b, T, T')$ records the allocation of two tags and an edge between them.

Two events pertain to executed actions: $\mathsf{exec}(i)$ records that i is executed, and $\mathsf{rf}(i, i')$ records both that i' is executed and i' read from i.

The final form, $\mathsf{co}(i, i')$, adds a coherence-order edge from i to i'. This is not used in the operational semantics, but it is useful in some proofs to have a way to add extraneous coherence edges.

Store transitions take two forms. There are synchronous transitions $H \overset{\delta@p}{\longmapsto} H'$, in which the δ is a transaction that is shared synchronously with a transition in thread p. Synchronous transitions handle the initiation of actions and the registration of new edges, as well as no-op transitions for empty and typing transactions. Asynchronous store transitions, $H \rightsquigarrow H'$ represent steps taken without any accompanying thread steps. It is these asynchronous store transitions that are responsible for, in the terminology of RMC, actually *executing* actions.

In the store transition rules, we write $H(\theta)$ to mean the event θ appears in H, where θ may contain wildcards $(*)$ indicating parts of the event that don't matter. As usual, we write $\rightarrow^+$ and $\rightarrow^*$ for the transitive, and the reflexive, transitive closures of a relation $\rightarrow$. We write the composition of two relations as $\overset{x}{\rightarrow}\overset{y}{\rightarrow}$. We say that $\rightarrow$ is *acyclic* if $\neg\exists x.\, x \rightarrow^+ x$.

Synchronous store transitions

We can give the rules for the synchronous transitions without any more preliminaries. Empty and typing transactions generate no new events. Edge transactions record the new edge, provided that the tags are distinct and fresh. (We define $\mathsf{tagdecl}_H(T)$ to mean $H(\mathsf{edge}(*, T, *)) \vee H(\mathsf{edge}(*, *, T))$.) An initiation transaction records the thread, the action, the value speculated, and any labels that apply, provided the identifier is fresh.

$\boxed{H \overset{\delta@p}{\longmapsto} H'}$

$$\frac{}{H \overset{\emptyset@p}{\longmapsto} H}\ (\textsc{None}) \qquad \frac{}{H \overset{m:\tau@p}{\longmapsto} H}\ (\textsc{Type})$$

$$\frac{\neg H(\mathsf{init}(i, *))}{H \overset{\vec{\varphi}\#i=\alpha\downarrow v@p}{\longmapsto} H, \mathsf{init}(i, p), \mathsf{is}(i, \alpha), \mathsf{spec}(i, v), [\mathsf{label}(\varphi, i) \mid \varphi \in \vec{\varphi}]}\ (\textsc{Init})$$

$$\frac{T \neq T' \quad \neg\mathsf{tagdecl}_H(T) \quad \neg\mathsf{tagdecl}_H(T')}{H \overset{T \overset{b}{\Rightarrow} T'@p}{\longmapsto} H, \mathsf{edge}(b, T, T')}\ (\textsc{Edge})$$

Store orderings

The general approach of weak memory models is to abandon any notion of "memory" and to replace it with a history that is given order by a soup of partial order relations. Before we can

give the rules for actually executing memory actions, we must present RMC's soup of predicates and ordering relations.

- Identifiers are in *program order* if they are initiated in order and on the same thread:

$$i \xrightarrow{\mathsf{po}}_H i' \stackrel{\text{def}}{=} \exists H_1, H_2, H_3, p.\ H = H_1, \mathsf{init}(i,p), H_2, \mathsf{init}(i',p), H_3$$

- An identifier is marked as executed by either an exec or an rf event:
$\mathsf{exec}_H(i) \stackrel{\text{def}}{=} H(\mathsf{exec}(i)) \vee H(\mathsf{rf}(*,i))$.
- *Trace order* is the order in which identifiers were actually executed:
$i \xrightarrow{\mathsf{to}}_H i' \stackrel{\text{def}}{=} \exists H_1, H_2.\ H = H_1, H_2 \wedge \mathsf{exec}_{H_1}(i) \wedge \neg\mathsf{exec}_{H_1}(i')$.
Note that this definition makes executed identifiers trace-order-earlier than non-yet-executed identifiers.
- *Specified order,* which realizes the tagging discipline, is defined by the rules:

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad H(\mathsf{label}(T,i)) \quad H(\mathsf{edge}(b,T,T')) \quad H(\mathsf{label}(T',i'))}{i \overset{b}{\Rightarrow}_H i'}$$

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad H(\mathsf{label}(\overset{b}{\lhd},i'))}{i \overset{b}{\Rightarrow}_H i'} \qquad \frac{i \xrightarrow{\mathsf{po}}_H i' \quad H(\mathsf{label}(\overset{b}{\rhd},i))}{i \overset{b}{\Rightarrow}_H i'}$$

- The key notion of *execution order* is defined in terms of specified order:
$i \xrightarrow{\mathsf{xo}}_H i' \stackrel{\text{def}}{=} \exists b.\, i \overset{b}{\Rightarrow}_H i'$.
- A read action is either a read or a read-write:
$\mathsf{reads}_H(i,\ell) = H(\mathsf{is}(i,\mathsf{R}[\ell])) \vee H(\mathsf{is}(i,\mathsf{RW}[\ell,*]))$.
- Similarly, a write action is either a write or a read-write:
$\mathsf{writes}_H(i,\ell,v) = H(\mathsf{is}(i,\mathsf{W}[\ell,v])) \vee H(\mathsf{is}(i,\mathsf{RW}[\ell,v]))$.
- An action i accesses some location l if it either reads or writes to l:
$\mathsf{accesses}_H(i,l) = \mathsf{reads}_H(i,l) \vee \mathsf{writes}_H(i,l,*)$
- Two accesses access the same location if they do:
$\mathsf{sameloc}_H(i,i') = \exists l.\mathsf{accesses}_H(i,l) \wedge \mathsf{accesses}_H(i',l)$
- *Reads-from:* $i \xrightarrow{\mathsf{rf}}_H i' \stackrel{\text{def}}{=} H(\mathsf{rf}(i,i'))$.
- An identifier is *executable* if (1) it has not been executed and (2) all its execution-order predecessors have been:

$$\mathsf{executable}_H(i) \stackrel{\text{def}}{=} \neg\mathsf{exec}_H(i) \wedge (\forall i'.\, i' \xrightarrow{\mathsf{xo}}_H i \supset \mathsf{exec}_H(i'))$$

- Identifiers i and i' are *push-ordered* if i is a push and is trace-order-earlier than i':
$i \xrightarrow{\pi\mathsf{o}}_H i' \stackrel{\text{def}}{=} H(\mathsf{is}(i,\mathsf{Push})) \wedge i \xrightarrow{\mathsf{to}}_H i'$
Since pushes are globally visible as soon as they execute, this means that i should be visible to i'.

- The key notion of *visibility order* is defined as the union of specified visibility order, reads-from, and push order: $i \xrightarrow{\mathsf{vo}}_H i' \stackrel{\text{def}}{=} i \xRightarrow{\mathsf{vis}}_H i' \vee i \xrightarrow{\mathsf{rf}}_H i' \vee i \xrightarrow{\pi\mathsf{o}}_H i'$.
- *Priority*: Priority is defined as the transitive closure of per-location program order and visible-to:

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad \mathsf{sameloc}_H(i, i')}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-PO)} \qquad \frac{i \xrightarrow{\mathsf{vt}}_H i' \quad \mathsf{sameloc}_H(i, i')}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-VT)}$$

$$\frac{i \xrightarrow{\mathsf{pri}}_H i'' \quad i'' \xrightarrow{\mathsf{pri}}_H i'}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-TRANS)}$$

  When $i \xrightarrow{\mathsf{pri}} i'$, we say that i is *prior to* i' or that i' "sees" i.

Coherence Order Finally, there is the central notion of coherence order (written $i \xrightarrow{\mathsf{co}}_H i'$), a strict partial order on writes to the same location that we introduced in Section 2.4.2.

Coherence order is the key mechanism by which writes a read can read from are restricted. The mechanism by which RMC's coherence order operates differs from most other operational models, such as Sarkar *et al.*'s [40]. Sarkar *et al.* takes an *ex ante* view of coherence order: coherence edges are introduced nondeterministically and asynchronously and must *already* be present for events to occur. In RMC, the coherence order is inferred *ex post* from events that have been executed; then, actions may only execute if their execution would not violate the key invariant of coherence order: that it is acyclic.

Intuitively, the first coherence rule states that a read always reads from the most recent (in coherence order) of all writes to its location that it has seen—or from some other more recent write:

$$\frac{i \xrightarrow{\mathsf{pri}}_H i_r \quad i' \xrightarrow{\mathsf{rf}}_H i_r \quad \mathsf{writes}_H(i, l, v) \quad i \neq i'}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-READ)}$$

Mechanically, this is accomplished by stating that if i_r reads-from i', all other writes that are prior to i_r must be coherence-earlier than i'. If i' is not the coherence-latest write that i_r has seen (or something coherence-later than it), then this would create a cycle in coherence order.

The second coherence rule states that writes are always more recent in coherence order than all other writes to its location that it has seen:

$$\frac{i \xrightarrow{\mathsf{pri}}_H i' \quad \mathsf{writes}_H(i, l, v) \quad \mathsf{writes}_H(i', l, v')}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-WRITE)}$$

The third coherence rule states that if a read-modify-write action i reads from a write i_w, no other write can come between them in coherence order:

$$\frac{i_w \xrightarrow{\mathsf{rf}}_H i \quad H(\mathsf{is}(i, \mathsf{RW}[*, *])) \quad i_w \xrightarrow{\mathsf{co}}_H i' \quad i \neq i'}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-RW)}$$

This is accomplished by requiring that every other write that is coherence-after i_w is also coherence-after i. If there was some write i' that was "in-between" i_w and i (that is, $i_w \xrightarrow{\mathsf{co}} i'$ and $i' \xrightarrow{\mathsf{co}} i$), then this rule would imply that $i \xrightarrow{\mathsf{co}} i'$, creating a cycle.

The final coherence rule is a technical device for some of the metatheory. It states that we can explicitly specify extra coherence edges in the history. The operational semantics will not introduce such events.

$$\frac{H(\mathsf{co}(i, i'))}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-FIAT)}$$

Asynchronous store transitions

$\boxed{H \rightsquigarrow H'}$

We can now give the asynchronous store transitions. These "execute" instructions in the terminology of RMC (though perhaps "resolve" would be a better term). There are two transitions: one for actions that perform a read and one for nonread actions. While all read actions and all nonread actions are treated the same for the purpose of execution, the presence of different actions in the history have very different effects.

For a read action i to be executed, it must have been initiated, it must be executable, there must be some already executed write to the same location to read from, the value of that write must agree with the speculated value of the read, and performing the read must not create a cycle in coherence order:

$$\frac{\begin{array}{l} H(\mathsf{spec}(i, v)) \quad \mathsf{reads}_H(i, \ell) \\ \mathsf{writes}_H(i_w, \ell, v) \quad \mathsf{exec}_H(i_w) \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H;\mathsf{rf}(i_w,i)}) \end{array}}{H \rightsquigarrow H, \mathsf{rf}(i_w, i)} \text{ (READ)}$$

The result of executing the read is to add $\mathsf{rf}(i_w, i)$ to the history. Note that there is nothing explicitly stating that the read must read the "right" value. The correctness of the read value comes entirely from the restriction that coherence order remain acyclic. If i_w is not a valid choice of write to read from, adding $\mathsf{rf}(i_w, i)$ to the history will create a cycle in coherence.

Executing a non-read is somewhat similar, though there are fewer moving pieces. To execute a write, a push, or a nop, it must have been initiated, have been speculated to return $()$, be executable, and not introduce a coherence cycle when executed.

$$\frac{\begin{array}{c} H(\mathsf{is}(i, \alpha)) \quad H(\mathsf{spec}(i, \texttt{()})) \\ \alpha = \mathsf{W}[\ell, v] \vee \alpha = \mathsf{Push} \vee \alpha = \mathsf{Nop} \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H;\mathsf{exec}(i)}) \end{array}}{H \rightsquigarrow H, \mathsf{exec}(i)} \text{ (NONREAD)}$$

The result of executing the action is to add $\mathsf{exec}(i)$ to the history.

3.1.5 Trace coherence

An important condition of the typing judgment for states is that a history be *trace coherent*. Trace coherence more or less states that a history could be generated by the dynamic semantics of

RMC. It checks, for example, that each identifier is initiated at most once, that every $\mathsf{is}(i, \alpha)$ event is immediately preceded by $\mathsf{init}(i, p)$, and that actions are only executed if they are executable. The one notable check is is missing is that coherence is acyclic, which is checked separately.

$\boxed{H\ \mathsf{trco}}$

$$\frac{}{\epsilon\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad \neg H(\mathsf{init}(i, *))}{H, \mathsf{init}(i, p)\ \mathsf{trco}} \qquad \frac{H\ \mathsf{trco} \quad H = H', \mathsf{init}(i, p)}{H, \mathsf{is}(i, \alpha)\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad H = H', \mathsf{is}(i, \alpha)}{H, \mathsf{spec}(i, v)\ \mathsf{trco}}$$

$$\frac{\begin{array}{l} H\ \mathsf{trco} \\ H(\mathsf{spec}(i_r, v)) \quad \mathsf{reads}_H(i_r, \ell) \\ \mathsf{writes}_H(i_w, \ell, v) \quad \mathsf{exec}_H(i_w) \\ \mathsf{executable}_H(i_r) \end{array}}{H, \mathsf{rf}(i_w, i_r)\ \mathsf{trco}} \qquad \frac{\begin{array}{l} H\ \mathsf{trco} \\ H(\mathsf{is}(i, \alpha)) \quad H(\mathsf{spec}(i, \ ()\ )) \\ \alpha = \mathsf{W}[\ell, v] \vee \alpha = \mathsf{Push} \vee \alpha = \mathsf{Nop} \\ \mathsf{executable}_H(i) \end{array}}{H, \mathsf{exec}(i)\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad T \neq T' \quad \neg\mathsf{tagdecl}_H(T) \quad \neg\mathsf{tagdecl}_H(T')}{H, \mathsf{edge}(b, T, T')\ \mathsf{trco}}$$

$$\frac{\begin{array}{l} H\ \mathsf{trco} \\ (\exists H', v.\, H = H', \mathsf{spec}(i, v)) \vee (\exists H', \varphi'.\, H = H', \mathsf{label}(\varphi', i)) \\ \mathsf{labeldecl}_H(\varphi) \end{array}}{H, \mathsf{label}(\varphi, i)\ \mathsf{trco}}$$

$$\frac{\mathsf{tagdecl}_H(T)}{\mathsf{labeldecl}_H(T)} \qquad \frac{}{\mathsf{labeldecl}_H(\overset{b}{\triangleleft})} \qquad \frac{}{\mathsf{labeldecl}_H(\overset{b}{\triangleright})}$$

$$\frac{H\ \mathsf{trco} \quad \mathsf{writes}_H(i, \ell, v) \quad \mathsf{writes}_H(i', \ell, v')}{H, \mathsf{co}(i, i')\ \mathsf{trco}}$$

3.1.6 Store static semantics

$$\text{signature} \quad \Sigma \quad ::= \quad (\Upsilon, \Phi)$$

$\boxed{\vdash \Upsilon\ \mathsf{ok} \quad \vdash H : \Upsilon}$

Tag signature checking, $\vdash \Upsilon\ \mathsf{ok}$, simply checks that all tags in Υ are distinct. Histories are checked against tag signatures via the judgment $\vdash H : \Upsilon$, which requires that Υ is well-formed and that tags appear in Υ if and only if they are declared in the history H.

$$\frac{}{\vdash \epsilon\ \mathsf{ok}} \qquad \frac{\vdash \Upsilon\ \mathsf{ok} \quad T \notin \Upsilon}{\vdash \Upsilon, T\ \mathsf{ok}}$$

$$\frac{\vdash \Upsilon\ \mathsf{ok} \quad \forall T.\, T \in \Upsilon \Leftrightarrow \mathsf{tagdecl}_H(T)}{\vdash H : \Upsilon}$$

$\boxed{\vdash \Phi\ \mathsf{ok} \quad \Upsilon; \Phi_0 \vdash H : \Phi}$

Location signature checking, $\vdash \Phi\ \mathsf{ok}$, simply checks that locations appear only once in the signature.

$$\frac{}{\vdash \epsilon\ \mathsf{ok}} \qquad \frac{\vdash \Phi\ \mathsf{ok} \quad \ell \notin \mathsf{Dom}(\Phi)}{\vdash \Phi, \ell{:}\tau\ \mathsf{ok}}$$

Checking a history against a location signature checks that all locations are initialized, that every value written is properly typed, and that all writes are to locations in the signature. The context that values are checked against is Φ_0 while Φ is the context being matched against the history H. The knot is tied by the top level history judgment, which will provide the same context for each.

$$\frac{\begin{array}{l}\vdash \Phi\ \mathsf{ok} \\ \forall(\ell{:}\tau){\in}\Phi.\, \mathsf{initialized}_H(\ell) \\ \forall(\ell{:}\tau){\in}\Phi.\, \forall i, v.\, \mathsf{writes}_H(i, \ell, v) \supset \Upsilon; \Phi_0; \epsilon \vdash v : \tau \\ \forall i, \ell, v.\, \mathsf{writes}_H(i, \ell, v) \supset \ell \in \mathsf{Dom}(\Phi)\end{array}}{\Upsilon; \Phi_0 \vdash H : \Phi}$$

The auxiliary definition $\mathsf{initialized}_H(\ell)$ says that there exists a write to ℓ that is visible to all reads from ℓ:

$$\begin{array}{l}\mathsf{initialized}_H(\ell) \stackrel{\mathrm{def}}{=} \\ \exists i_w, i_p, v.\, \mathsf{writes}_H(i_w, \ell, v) \wedge H(\mathsf{is}(i_p, \mathsf{Push})) \\ \qquad\qquad \wedge\, i_w \stackrel{\mathsf{vo}}{\rightarrow}{}^{+}_H\ i_p \wedge \mathsf{exec}_H(i_p) \\ \qquad\qquad \wedge \forall i_r.\, \mathsf{reads}_H(i_r, \ell) \supset i_p \stackrel{\mathsf{to}}{\rightarrow}_H i_r\end{array}$$

$\boxed{\vdash H : \Sigma}$

The top level history checking judgment, $\vdash H : \Sigma$, checks a history against a signature $\Sigma = (\Upsilon, \Phi)$. We require that H be trace coherent, coherence-acyclic, that its tags match the tag signature Υ, and that it checks against the location signature Φ.

$$\frac{\begin{array}{cc}H\ \mathsf{trco} & \mathsf{acyclic}(\stackrel{\mathsf{co}}{\rightarrow}_H) \\ \vdash H : \Upsilon & \Upsilon; \Phi \vdash H : \Phi\end{array}}{\vdash H : (\Upsilon, \Phi)}$$

3.1.7 Signature dynamic semantics

$\boxed{\Sigma \stackrel{\delta@p}{\longmapsto} \Sigma'}$

Like history and expression stepping, signature stepping synchronizes on a transaction. The stepping rules for typing and initiation transactions make no changes to the signature, but merely check that that everything is well formed.

$$\frac{\Upsilon, \Phi, \epsilon \vdash m : \tau}{(\Upsilon, \Phi) \overset{m:\tau@p}{\longmapsto} (\Upsilon, \Phi)} \qquad \frac{\begin{array}{c}\forall \varphi \in \vec{\varphi}.\ \Upsilon; \epsilon \vdash \varphi : \mathsf{label} \\ \Upsilon, \Phi, \epsilon \vdash \alpha : \tau \quad \Upsilon, \Phi, \epsilon \vdash v : \tau\end{array}}{(\Upsilon, \Phi) \overset{\vec{\varphi}\#i=\alpha\downarrow v@p}{\longmapsto} (\Upsilon, \Phi)}$$

$$\frac{}{\Sigma \overset{\emptyset@p}{\longmapsto} \Sigma}$$

The edge creation transaction checks that the tags are distinct and unique, and adds them to the signature.

$$\frac{T \neq T' \quad T, T' \notin \Upsilon}{(\Upsilon, \Phi) \overset{T \overset{b}{\Rightarrow} T'@p}{\longmapsto} ((\Upsilon, T, T'), \Phi)}$$

3.1.8 Top-level semantics

We now have all of the pieces, and can put everything together. The top-level state of RMC consists of the ambient signatures, the history, and the execution state.

$$\text{state} \quad s \quad ::= \quad (\Sigma, H, \xi)$$

$\boxed{\vdash s\ \mathsf{ok}}$

Top-level state checking checks the history and the execution state against the signature.

$$\frac{\vdash H : (\Upsilon, \Phi) \quad \Upsilon, \Phi \vdash \xi\ \mathsf{ok}}{\vdash ((\Upsilon, \Phi), H, \xi)\ \mathsf{ok}}$$

$\boxed{s \mapsto s'}$

States take a synchronous step when the signature, history, and thread execution state all take a step that synchronizes on the same transaction. States also step when the history makes an asynchronous step.

$$\frac{\Sigma \overset{\delta@p}{\longmapsto} \Sigma' \quad H \overset{\delta@p}{\longmapsto} H' \quad \xi \overset{\delta@p}{\longmapsto} \xi'}{(\Sigma, H, \xi) \mapsto (\Sigma', H', \xi')} \qquad \frac{H \rightsquigarrow H'}{(\Sigma, H, \xi) \mapsto (\Sigma, H', \xi)}$$

3.2 Discussion

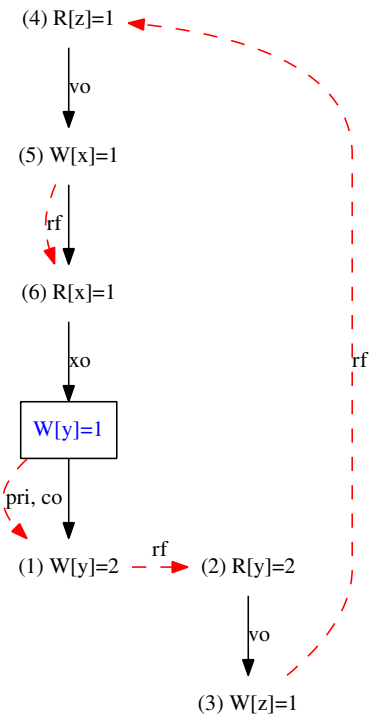
3.2.1 Mootness, incorrect speculation, and semantic deadlock

One consequence of the highly nondeterministic semantics of RMC and the loose semantics of stores is that programs can find themselves backed into dead ends in which it is impossible to execute some or all of the initiated actions in the history (though, because of how loosely the thread execution is tethered to the store execution, threads will keep stepping even if no progress can be made actually executing the actions they initiate!).

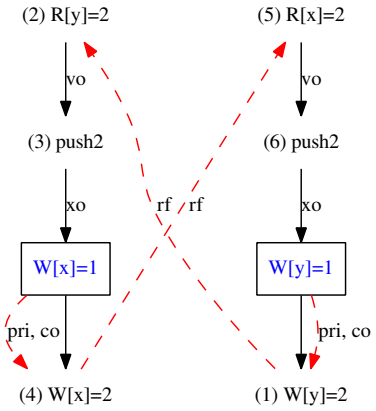
There are two ways in which execution can find itself in such a stuck state, though both boil down to inconsistent choices having been made by the nondeterministic semantics. The simplest way is simply via incorrect speculation. When a read is initiated, it needs to choose what value it is going to read; for the read to execute, it needs to be able to actually read that value. If the value does not match any write that the read can or will be able to read from, the read will never be able to execute.

The second way this can occur is through what we call *semantic deadlock*. This is a different phenomenon than ordinary deadlock, in which a buggy program can not make progress due to unbreakable circularities in its resource acquisition protocol. In *semantic* deadlock, a (potentially, at least) correct program is unable to make progress because any progress would create a cycle in coherence order. This can be caused by either reads or pushes.

As an example of it being caused by reads, consider the following execution graph:



If the boxed action is removed from this graph (and considered to be initiated but not yet executed), it is a perfectly valid execution state reachable by executing the unboxed actions in the order indicated by their labels. The write `W[y]=2` has been executed before a program-order-earlier conflicting write, but that is totally allowed. However, in this situation, we cannot execute `W[y]=1` without creating a cycle in coherence order. This is because `W[y]=2` $\xrightarrow{vo}{}^{+}$ `R[x]=1` and thus `W[y]=2` $\xrightarrow{vt}$ `W[y]=1` and `W[y]=2` $\xrightarrow{co}$ `W[y]=1`. Thus this state is *semantically deadlocked*. The problem can also arise with pushes as follows:



Here, we again consider the unboxed actions to have executed and the boxed ones to have merely been initiated. The labels on the actions show one possible order the actions may have been executed, though there are others: 1–3 and 4–6 form independent chains that can be interleaved with each other. Here, neither `W[x]=1` nor `W[y]=1` may be executed. If `W[y]=1` executes, since `push1` is trace-order before it, `push1` $\xrightarrow{\mathsf{vo}}$ `W[y]=1` and thus `W[y]=2` $\xrightarrow{\mathsf{vt}}$ `W[y]=1`, which would cause a cycle in coherence order. Similar logic applies for `W[x]=1`.

We refer to incorrect speculation and semantic deadlock together as *mootness* and say that a state with unexecutable actions is *moot*.

Formally, we say that a state is moot if no state that it can evaluate to is *complete* (every initiated action has been executed). This version of the definition is unfortunately a little overbroad—in addition to incorrect speculation and semantic deadlock, it also could include incorrect *stuck* states caused by a failure of type safety. These states, of course, should not be possible—see Section 3.3.1 for a discussion of this.

Mootness can not actually occur in practice when executing programs, so we generally restrict our consideration to states that are not moot.

s complete

$$\frac{\forall i, p.\, H(\mathsf{init}(i,p)) \supset \mathsf{exec}_H(i)}{(\Sigma, H, \xi)\ \mathsf{complete}}$$

s moot

$$\frac{\nexists s'.\, s \mapsto^* s' \wedge s'\ \mathsf{complete}}{s\ \mathsf{moot}}$$

Some important properties of mootness include:

1. If s moot and $s \mapsto^* s'$, then s' moot.
2. If $(\Sigma, H, \xi \parallel p : e)$ moot, then (Σ, H, ξ) moot.
3. If (Σ, H, ξ) cannot take a step, and there are unexecuted actions in H, then (Σ, H, ξ) is moot.
4. If every reduct of s is moot, then s moot.

3.2.2 Read-read coherence

Unlike most other weak memory models, early versions of RMC did not have read-read coherence; that is, two program-order reads from the same location would not necessarily agree with coherence order.

For RMC 2.0, we decided to add read-read coherence to RMC. The main reasons for this is that it slightly simplifies the definition of priority and that it was an opportunity to reduce surprises to people used to other memory models at very low cost. Nearly all extant hardware architectures provide read-read coherence, as does the C++11 memory model for relaxed accesses. The primary relevant exception to this is the Itanium architecture—while we do not currently support Itanium, we believe that we could by following C++'s example and compiling all atomic accesses with Itanium's release and acquire instructions [27**?**].

There are some more concrete advantages, however. Consider the following code fragment:

```
do {
   while (flag != 0) continue;
} while (!flag.compare_and_swap(0, 1));
```

Here, we have a test-and-test-and-set loop: we spin, waiting for a flag to become zero. Once it is zero, we attempt to swap it with one. However, even if the initial test exits, without read-read coherence the test-and-set might read a coherence-prior read and fail, even in the absence of another thread making progress. While in this case the consequences of such would be minor (just more times through the loop), many lock-free algorithms depend on the knowledge that a compare-and-swap failing implies that another thread made progress in order to achieve lock-freedom. It is also possible to construct examples where correctness more explicitly relies on this. While it is possible to repair these concerns with execution edges, that seems like perhaps overkill to achieve guarantees already provided by the hardware.

The downside of requiring read-read coherence is that we can no longer give the same pithy and elegant rationale for the design of coherence that we did previously: that coherence is as weak as possible subject to the constraint that message-passing should work using visibility and execution order and that single-threaded programs must work as expected.

If we wished to switch back to not promising read-read coherence, we could simply change the program-order priority rule to only apply to conflicting accesses (so that it would cover read-write, write-read, and write-write):

- Two actions conflict if they access the same location and at least one is a write:
  $\mathsf{conflict}_H(i, i') = \exists l, v.\mathsf{sameloc}_H(i, i') \wedge (\mathsf{writes}_H(i, l, v) \vee \mathsf{writes}_H(i', l, v))$

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad \mathsf{conflict}_H(i, i')}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-PO)}$$

3.2.3 Connections to RMC-C++

At first glance, the core RMC language does not seem to be a very good match for RMC-C++ at all. After all, RMC-C++ is essentially a traditional imperative language while RMC is a monadic lambda calculus. These things are not as different as they appear, however, and it is

fairly straightforward to map RMC-C++ to core RMC. We will not present anything like a full translation from C++ to RMC, but we will touch on the most salient issues.

The RMC core calculus is intentionally very spare, to avoid distracting from the core issues. In our discussion of how RMC-C++ could be mapped to the RMC core calculus, we freely assume that the term language of RMC is extended with various standard features such as let bindings, tuples, and mutual recursion.

From C++ to a monadic lambda calculus

While a monadic lambda calculus seems quite different than a traditional block structured imperative programming, it is fairly simple to translate from a traditional imperative language to this sort of calculus. We will not go into detail about this sort of translation, except to touch on some details. Because C++ expressions can do memory operations, both C++ expressions and C++ statements are mapped to RMC expressions. This requires expressions to be linearized into a sequence of operations. Looping constructs may be implemented using recursion. Mutable local variables (assignables) require some method of allocation, but once we add that in Section 3.4.7, they pose no problem.

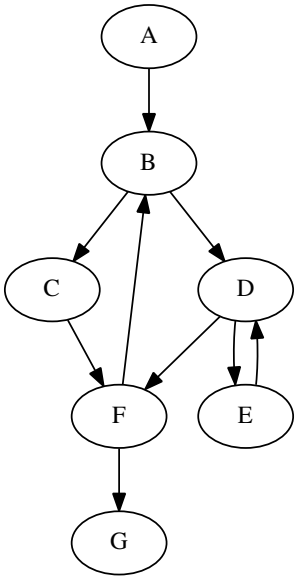
Edges and labels

There is a slight mismatch between how labeling works in RMC-C++ and the RMC core calculus. In the RMC calculus, tags are bound by the edge declaration form and each tag has exactly one partner. The more flexible RMC-C++ method can be transformed into this by using label names to determine which tags to apply to each expression. Then a declaration `VEDGE(foo, bar)` would be transformed into an expression $\mathsf{new}\ t_f \overset{\mathsf{vis}}{\Rightarrow} t_b.e$, where e is the translation of the scope of the declaration (as discussed below). Inside e, every action that had been tagged with `foo` will be tagged with t_f and likewise for t_b. If `foo` or `bar` are used in multiple edge declarations, all of their actions will receive multiple tags.

Edge binding structure

The fiddliest bit of interpreting RMC-C++ through the lens of the RMC core calculus is in handling binding sites for edges. In RMC-C++, the scope of an `?EDGE_HERE` declaration is defined to be all of the basic blocks that are dominated in the control flow graph by the declaration site for the function while in the RMC formalism the scope of a edge declaration is essentially standard lambda calculus variable scoping.

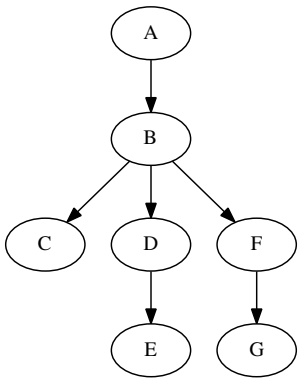
Fortunately, it is possible to convert a control flow graph to a lambda calculus program in such a way that the binding structure mirrors the structure of dominance in the control flow graph. As an example, consider this example control flow graph:



The most straightforward way of compiling a basic block structured program like this to a lambda calculus is to mutually recursively define one function per basic block. Transferring control between blocks is then done by making a tail-call to the target block. We will demonstrate the general approach by sketching out what the compiled program would look like in an ML-style target language:

```
fun func ... = let
   fun A () = (* A body *)
   and B () = (* B body *)
   and C () = (* C body *)
   and D () = (* D body *)
   and E () = (* E body *)
   and F () = (* F body *)
   and G () = (* G body *)
 in A () end
```

This works as far as representing the control flow goes, but since the binding structure does not match the structure of the control flow dominance it is not suitable for representing RMC-C++'s edge binding. The key insight for representing dominance relationships in the binding structure is to take advantage of the notion of *immediate dominators* and its representation as a *dominator tree*. Recall that A *dominates* B if every path from the entry to B passes through A. This means that nodes dominate themselves, so we say that A *strictly dominates* B if A dominates B and $A \neq B$. We then say that A *immediately dominates* B if A strictly dominates B and there does not exist some C such that A strictly dominates C and C strictly dominates B. That is, A is B's immediate dominator iff there is no element in between A and B in the strict dominance partial order. Immediate dominance can be used to create a dominator tree, in which a node's children are all of the nodes that it immediately dominates. The dominator tree for the above example control flow graph (CFG) is:



The dominator tree is a very convenient tool for viewing dominance. One node dominates another if and only if it is an ancestor in the dominator tree.

It turns out, then, that it is fairly straightforward to rework the binding structure such that it mirrors the dominance tree. Instead of binding all basic blocks in one big mutually recursive binding, each basic block (mutually recursively) binds all of the blocks that it immediately dominates, in the scope of its body:

```
fun func ... = let
   fun A () = let
      fun B () = let
         fun C () = (* C body *)
         and D () = let
            fun E () = (* E body *)
            in (* D body *) end
         and F () = let
            fun G () = (* G body *)
            in (* F body *) end
         in (* B body *) end
      in (* A body *) end
   in A () end
```

Now the binding structure mirrors the dominance structure of a program: a node B's definition is nested inside of the definition of A if and only if A dominates B. As a consequence, any definition that appears in a block A is visible to all blocks dominated by A. The main remaining question is: does this work?

Specifically, we require that if a block A can jump to B, B is bound in A's body (so that A can actually call B!). In our scheme, a block B's binding is visible in A if A is immediately dominated by some block D and D dominates B—since B is immediately dominated by D, it is bound in D and therefore visible to the body of B. Thus we require that if there is a control flow edge from A to B, B's immediate dominator D must dominate A. Fortunately, this is fairly trivial: suppose that D did not dominate A: then there is some path from the entry point to A that does not pass through D; but then there is also a path from the entry, through A, to B, that does not pass through D, a contradiction.

Give and take

The intuition that we gave for RMC-C++ style `LGIVE` and `LTAKE` is that the act of passing an argument or returning a value can be considered as an *action* and that `LGIVE` and `LTAKE` provide a way to name that action. Of course, passing an argument or returning a value is not actually an action; it is merely the basic operation of the lambda-calculus style term language. It turns out, however, that it is fairly easy to *simulate* such an action. To do this, we wrap each function argument and return value in a suspension. Each function argument and return value that would be of type τ is instead, under this scheme, of type τ susp; this can be thought of a suspended expression that, when executed, "passes" the value. More concretely, it will execute a (tagged) Nop and then return a value.

On the `LGIVE` side of things, a suspended transfer of v tagged with $\vec{\varphi}$ can be created with $(\mathsf{susp}\,(\_ \leftarrow \vec{\varphi}\,\#\,\mathsf{Nop\ in\ ret}\,v))$. On the `LTAKE` side, a suspended action m can be given additional tags and then have its value extracted by sequencing $(\varphi\,\#\,\mathsf{force}\,m)$. Then, when a function call is made, the caller creates transfer actions for each argument and, if an argument is an `LGIVE`, tags the argument with any matching tags. On the callee side, functions tags any `LTAKE` actions with the appropriate tag, and then executes them.

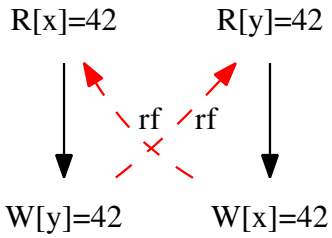
From the perspective of the formalism, then, there is nothing special about the "argument" or any direct relationship to data dependency. `LGIVE` and `LTAKE` can then be viewed as a combination of a convention about argument passing combined with a heuristic for generating code that uses them.

3.2.4 Thin-Air Reads

Unfortunately, RMC as formalized permits bizarre "out-of-thin-air" results. Thin-air results are executions in which the value returned from a read is used to justify the write that is being read from. The canonical example of this is:

```
r1 = x;     r2 = y;
y = r1;     x = r1;
```

RMC permits the bizarre outcome of `r1 == r2 == 42`. From the RMC perspective, this can occur like this: the first thread initiates a read from `x`, speculating that it will return 42. It then initiates a write to `y` with the value 42. This write can then be executed (out-of-order), before the read. Thread two then can read `r2 = 42` and thus write `x = 42`. Once that latter write has been executed, we can execute the thread 1 read, which reads from a write of 42 to `x`, thus satisfying the speculation. As a event diagram, this looks like:



This sort of behavior is bizarre and certainly has the possibility of breaking abstraction guarantees, but is perhaps survivable. Boehm and Demsky [13] demonstrated that the situation is actually much worse:

```
struct nobe {
   atomic<struct nobe *> next;
};
struct nobe *foo, *bar;

r1 = foo->next;    r2 = bar->next;
r1->next = foo;    r2->next = bar;
```

In this example, two threads each attempt to create a loop in a linked list. If the lists `foo` and `bar` are disjoint, by any reasonable interpretation of how computer programs ought to work, the two threads ought to have no interaction. Unfortunately, without some way to disallow out-of-thin-air executions, this trace admits the outcome `r1 == bar, r2 == foo`. Thus we set `bar->next = foo` and `foo->next = bar`, spontaneously linking together two previously-unrelated data structures being operated on by entirely innocent code! It is basically impossible to reason about programs when this sort of abomination is permitted. Furthermore, Vafeiadis et al. [47] demonstrated that the thin-air problem renders several common compiler transformations invalid in the C11 memory model.

Hardware memory models easily side step this problem, because there is a data dependency that prevents reordering. Language memory models can not promise to respect dependencies in such a way without throwing the baby out with the bathwater and prohibiting useful optimizations as well as the undesired behaviors.

The thin-air problem has a sordid history. The most complex part of Java's memory model [32] was intended to rule out thin-air behaviors without banning common optimizations; this was unsuccessful [48]. The original C++ memory model contained a half-hearted attempt to rule out thin-air, but it had semantics that were much too strong; C++14 instead contains a hand-wavy requirement that implementations not allow out-of-thin-air values. Vafeiadis et al. [47] referred to the search for an efficiently implementable model without thin-air behaviors as a "Holy Grail quest". Fortunately, shortly thereafter that Kang et al. [29]'s "Promising Semantics for Relaxed-Memory Concurrency" seems to have recovered the grail.

A potential approach

Unfortunately, while we believe that it is possible to apply the key ideas of Kang et al. [29] to RMC, we have not yet done so. A strawman proposal for applying these ideas, at a high level, looks something like:

- A thread p may *promise* to write a value v to a location l, if it can show that there is an execution in which it will in fact perform that write *with no undischarged read speculations by p in its storage history*.
- A speculated write event can only be executed if there is a pre-existing promise to perform that write.
- Execution must preserve the satisfiability of all outstanding promises.

With a proper formalization of a solution to thin-air reads as yet incomplete for RMC, we content ourselves for now with the terribly handwavy prose side-condition used by C++: "Implementations should ensure that no 'out-of-thin-air' values are computed that circularly depend on their own computation." [28].

3.3 Metatheory

The metatheory discussed here was formalized in Coq for RMC 1.0 by Karl Crary and updated to RMC 2.0 by Joseph Tassarotti. It has not been extended with the extensions discussed later in this chapter.

3.3.1 Type safety

Type safety is normally given as a combination of a preservation and progress theorem. The preservation theorem in RMC is fairly straightforward:

Theorem 1 (Preservation)**.** *If* $s \mapsto s'$ *and* $\vdash s\ \mathsf{ok}$ *then* $\vdash s'\ \mathsf{ok}$.

In our setting, however, the traditional statement of progress (which would be something like if $\vdash s\ \mathsf{ok}$ then either $s \mapsto s'$ or s is final) is unsatisfying. In fact, it is both too strong and too weak. To see that it is too weak, note that it is hypothetically possible for one thread to be stuck in a bad state while another thread can still step. In fact, because of speculation, it would often (though not always) be possible for a thread to step even while in a bad state.

On the flip side, because of RMC's approach to nondeterminism, even perfectly well behaved programs can become stuck due to incorrect speculation or semantic deadlock (as discussed in Section 3.2.1). These states are stuck, but they are not "bad". They are just the results of making the wrong nondeterministic choices.

Our approach then is to explicitly characterize "good" RMC states with a judgment $s\ \mathsf{right}$ (the details of which are elided in this document). Importantly, $s\ \mathsf{right}$ draws the distinction between valid stuck states that result from wrong nondeterministic choices and bad stuck states that result from the program "going wrong". We then can prove:

Theorem 2 (Rightness)**.** *If* $\vdash s\ \mathsf{ok}$ *then* $s\ \mathsf{right}$.

This is similar to Wright and Felleisen's original technique [49], except they characterized the faulty states instead of the good ones, and for them faultiness was a technical device, not the notion of primary interest.

We can then connect rightness back to the thread side of the operational semantics—because mootness only prevents execution of heap transitions, we can still prove a progress-like theorem about thread execution:

Theorem 3 (Progress)**.** *If* $(\Sigma, H, \xi)\ \mathsf{right}$*, then for every thread* $p : e$ *in* ξ*, either (1) there exist* $\delta, \Sigma', H', \xi'$ *such that* $\Sigma \overset{\delta@p}{\longmapsto} \Sigma'$*,* $H \overset{\delta@p}{\longmapsto} H'$*, and* $\xi \overset{\delta@p}{\longmapsto} \xi'$ *(that is, the thread* p *steps), or (2)* e *is of the form* $\mathsf{ret}\, v$.

3.3.2 Sequential consistency results

We present two theorems about RMC showing that, under certain programming disciplines, sequentially consistent behavior may be recovered. First, we must define sequential consistency in the setting of RMC.

Definition 4. Suppose R is a binary relation over identifiers and S is a set of identifiers. Then R is a *sequentially consistent ordering for* S if (1) R is a total order, (2) R respects program order for elements of S, and (3) every read in the domain of R is satisfied by the most R-recent write to its location.

If there exists a sequentially consistent ordering over *all* memory operations, then the entire program can be said to be sequentially consistent. There being a sequentially consistent ordering on even a subset of actions can still be useful, however.

One more definition is needed to state the theorems:

Definition 5. A history H is *consistent* if there exists H' containing no is events, such that H, H' is trace coherent, complete, and coherence acyclic.

We may now present the theorems. We elide the proofs, which appear in the original paper [18].

Pushes recover sequential consistency

Our first sequential consistency results says that sequential consistency may be recovered by inserting pushes between all actions. We define *specified sequential consistency* to mean that two actions are separated by a push:

$$i \overset{\mathsf{ssc}}{\to}_H i' \quad \overset{\text{def}}{=} \quad \exists i_p.\, H(\mathsf{is}(i_p, \mathsf{Push})) \wedge i \overset{\mathsf{vo}}{\to}^+_H i_p \overset{\mathsf{xo}}{\to}^+_H i'$$

Theorem 6 (Sequential consistency)**.** *Suppose H is consistent. Suppose further that S is a set of identifiers such that for all $i, i' \in S$, $i \overset{\mathsf{po}}{\to}_H i'$ implies $i \overset{\mathsf{ssc}}{\to}_H i'$. Then there exists a sequentially consistent ordering for S.*

Observe that the sequentially consistent order includes *all* of H's writes, not only those belonging to S. Thus, writes not belonging to S are adopted into the sequentially consistent order. This means that the members of S have a consistent view of the order in which *all* writes took place, not just the writes belonging to S. However, for non-members that order might not agree with program order. (The order contains non-S reads too, but for purposes of reasoning about S we can ignore them. Only the writes might satisfy a read in S.)

Data-race-freedom and sequential consistency

Our second theorem roughly states that data-race-free executions are sequentially consistent. In order to state this, we first need an appropriate definition of data-race-free. To do so, we define a *synchronized-before* relation:

$$i \overset{\mathsf{sb}}{\to}_H i' \quad \overset{\text{def}}{=} \quad \exists i_s.\, i \overset{\mathsf{vo}}{\to}^+_H i_s \overset{\mathsf{po}}{\to}^*_H i' \wedge H(\mathsf{label}(\overset{\mathsf{exe}}{\rhd}, i_s))$$

In essence $i \overset{\mathsf{sb}}{\to} i'$ says that i is visible before some synchronization operation i_s that is execution-order before i'. That i_s must be labeled $\overset{\mathsf{exe}}{\rhd}$ so that i will also be synchronized before any program-order successors of i'.

We say that a set of identifiers from an execution is data-race-free if any two conflicting actions in the set on different threads are ordered by synchronized-before. Boehm and Adve [12] refer to this sort of data race as a "type 2 data race." We may then show that data-race-free executions are sequentially consistent:

Theorem 7 (Sequential consistency)**.** *Suppose H is consistent. Suppose further that S is a data-race-free set of identifiers from H. Then there exists a sequentially consistent ordering for S.*

A straightforward corollary of this is that if *every* possible execution of a program is data-race-free, then every execution is sequentially consistent as well.

Note that this theorem is able to apply to data-race-free fragments of an execution even if the entire execution is not data-race-free. This is important because it allows the synchronization mechanisms themselves to contain data races without compromising the sequential consistency of client code. This result is thus insensitive to the mechanism of synchronization; it might use atomic compare-and-swaps, the Bakery algorithm, or other approaches.

3.4 Improvements and new features

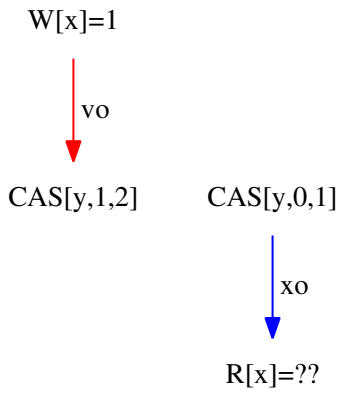
Of the improvements in this section, push edges and thread spawning come from RMC 2.0 [19] while the others are new work.

3.4.1 Better compare-and-swap

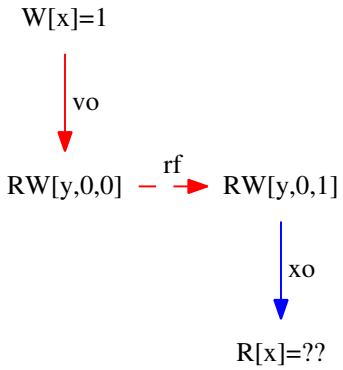
Basic RMC provides a generic RMW action for performing atomic read-modify-writes. While most atomic read-modify-writes can cleanly be expressed in terms of it, an implementation of compare-and-swap using it is problematic. There is an obvious and tempting potential implementation of compare-and-swap using RMW:

$$\mathsf{CAS}[p, e, n] = \mathsf{RMW}[p, x.\ \mathsf{if}\ x = e\ \mathsf{then}\ n\ \mathsf{else}\ x]$$

In this implementation, we atomically check to see if the value at the location matches our expected value e and write in our new value n; if it doesn't, we write back the value x that was already there. It's this last step that is the heart of the problem: even when a compare-and-swap fails it still performs a write, according to RMC. This write can then induce changes in coherence order that constrain the observable behavior. This is unfortunate, since the natural implementation of compare-and-swap for ARM and POWER does *not* perform a write in this case. To see how this can be a problem, consider the following program:

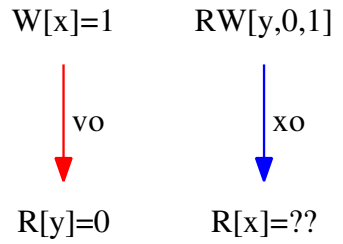


Here, thread one writes a value to x, visibility-ordered before trying to swap y from 1 to 2. Thread two meanwhile tries to swap y from 0 to 1, execution-ordered before reading from x. If CAS is implemented with RMW, the case in which thread one's CAS fails looks like this:



Read-writes to the same location are totally ordered, and because thread one does not see thread two's write to y, thread one's no-op write must be read by thread two. This implies that thread two must read $x = 1$.

Unfortunately this does not match how we want to compile for POWER and ARM. Compare-and-swap on those architectures is implemented using load-linked and store-conditional facilities in such a way that if the compare fails, no write is done at all. This mismatch with RMC is the source of the problem. This implementation means that the trace actually would correspond to:



This trace, and its concrete realizations on ARM and POWER, does *not* guarantee that the read will read 1. The problem here is this trace is essentially the store-buffering trace: two threads

each perform a write and we want to ensure that at least one of the thread's observes the other write. Unfortunately, as discussed earlier, this requires a push on each side.

Having the equivalent of a push in the generated code for compare-and-swap would be unfortunate: it would mean that visibility edges into CASes and execution edges out of CASes must both be implemented with a full `sync` on POWER (and on ARM both sides would need to be a `dmb` where otherwise the execution side could be done with dependencies).

Fixing it

The cause of these unnecessarily strict semantics for failed compare-and-swaps is the requirement in Basic RMC that all RMW operations perform a write, even though it does not really make sense for a failed compare and swap to do so. We can fix this by making the write *optional*: we can add another field to RMW that is used to decide whether a write should be done. Like the subterm that decides what value to write back, this subterm binds a variable containing the old value; it then evaluates to a natural number (since the threadbare RMC core calculus lacks booleans) that determines whether to perform a write.

$$\begin{array}{rrcl} \text{expr's} & e & ::= & \mathsf{RMW}[m, x{:}\tau.m, x{:}\tau.m] \\ & & \mid & \mathsf{RMWS}[\ell, v, m, m] \end{array}$$

Statics:

$$\frac{\Upsilon;\Phi;\Gamma \vdash m_1 : \tau\ \mathsf{ref} \quad \Upsilon;\Phi;\Gamma, x{:}\tau \vdash m_2 : \mathsf{nat} \quad \Upsilon;\Phi;\Gamma, x{:}\tau \vdash m_3 : \tau}{\Upsilon;\Phi;\Gamma \vdash \mathsf{RMW}[m_1, x{:}\tau.m_2, x{:}\tau.m_3] : \tau}$$

$$\frac{\Upsilon;\Phi;\Gamma \vdash \ell : \tau\ \mathsf{ref} \quad \Upsilon;\Phi;\vdash v : \tau \quad \Upsilon;\Phi;\vdash m_1 : \mathsf{nat} \quad \Upsilon;\Phi;\vdash m_2 : \tau}{\Upsilon;\Phi;\Gamma \vdash \mathsf{RMWS}[\ell, v, m_1, m_2] : \tau}$$

Dynamics:

$$\frac{m_1 \mapsto m_1'}{\mathsf{RMW}[m_1, x{:}\tau.m_2, x{:}\tau.m_3] \overset{\emptyset}{\mapsto} \mathsf{RMW}[m_1', x{:}\tau.m_2, x{:}\tau.m_3]}$$

$$\frac{}{\mathsf{RMW}[\ell, x{:}\tau.m_b, x{:}\tau.m] \overset{v:\tau}{\longmapsto} \mathsf{RMWS}[\ell, v, [v/x]m_b, [v/x]m]}$$

$$\frac{m_b \mapsto m_b'}{\mathsf{RMWS}[\ell, v, m_b, m] \overset{\emptyset}{\mapsto} \mathsf{RMWS}[\ell, v, m_b', m]}$$

$$\frac{m \mapsto m'}{\mathsf{RMWS}[\ell, v, s(v_n), m] \overset{\emptyset}{\mapsto} \mathsf{RMWS}[\ell, v, s(v_n), m']}$$

$$\frac{}{\mathsf{RMWS}[\ell, v_s, 0, m] \overset{\epsilon \# i = \mathsf{R}[l]\downarrow v_s}{\longmapsto} \mathsf{ret}\ v_s}$$

$$\frac{}{\mathsf{RMWS}[\ell, v_s, s(v_b), v_m] \overset{\epsilon \# i = \mathsf{RW}[l, v_m]\downarrow v_s}{\longmapsto} \mathsf{ret}\ v_s}$$

3.4.2 Push edges

In section 2.1.5, we introduced push actions and a derived notion of "push edges". We wish to properly support push edges in our calculus. To support this syntactically, we add a push attribute that may be given as an edge type.

$$\text{attributes} \quad b \quad ::= \quad \cdots \mid \mathsf{push}$$

The intuition of a push edge is that given $i' \overset{\mathsf{push}}{\Rightarrow} i$, a push action i_p will be inserted such that $i' \overset{\mathsf{vo}}{\to} i_p$ and $i_p \overset{\mathsf{xo}}{\to} i$. To realize this, we will allow threads to spontaneously generate push actions that are visibility-after an arbitrary set of other actions on the thread.

To track this visibility ordering, we add an event $\mathsf{spo}(i, i')$ to record that i is visibility-before a spontaneously generated push i' and we extend the definition of visibility order to include this.

- events $\theta \quad ::= \quad \cdots \mid \mathsf{spo}(i, i)$
- *Spontaneous-push-order* means that an identifier is to be visibility-before a spontaneously generated push: $i \overset{\mathsf{spo}}{\to}_H i' \overset{\text{def}}{=} H(\mathsf{spo}(i, i'))$.
- $i \overset{\mathsf{vo}}{\to}_H i' \overset{\text{def}}{=} i \overset{\mathsf{vis}}{\Rightarrow}_H i' \vee i \overset{\mathsf{rf}}{\to}_H i' \vee i \overset{\pi\mathsf{o}}{\to}_H i' \vee i \overset{\mathsf{spo}}{\to}_H i'$.

We then need to extend the definition of executable to require that (1) all of the identifier's spontaneous-push-order predecessors are executed and (2) all of the identifier's push-order predecessors are spontaneous-push-ordered before a push that has been executed:

$$\begin{aligned}\mathsf{executable}_H(i) \overset{\text{def}}{=} \neg\mathsf{exec}_H(i) \wedge (\forall i'.\, i' \overset{\mathsf{xo}}{\to}_H i \supset \mathsf{exec}_H(i')) \\ \wedge (\forall i'.\, i' \overset{\mathsf{spo}}{\to}_H i \supset \mathsf{exec}_H(i')) \\ \wedge (\forall i'.\, i' \overset{\mathsf{push}}{\Rightarrow}_H i \supset \exists i_p.\, H(\mathsf{is}(i_p, \mathsf{Push})) \wedge \mathsf{exec}_H(i_p) \wedge i' \overset{\mathsf{spo}}{\to}_H i_p)\end{aligned}$$

This does not *quite* accomplish exactly the intuition given above for $i' \overset{\mathsf{push}}{\Rightarrow} i$. We will have that $i' \overset{\mathsf{vo}}{\to} i_p$ for a push i_p, but we will not literally have that $i_p \overset{\mathsf{xo}}{\to} i$, since $\overset{\mathsf{xo}}{\to}$ is only defined in terms of specified order. However, while we don't literally have $i_p \overset{\mathsf{xo}}{\to} i$, we do have that $i_p \to i$ and so $i_p \overset{\mathsf{vo}}{\to} i$. Since i_p is spontaneously generated, we will only have $i'' \overset{\mathsf{xo}}{\to} i_p$ if i'' has a post edge; this means that any $\overset{\mathsf{vt}}{\to}$ edges that would be present if $i_p \overset{\mathsf{xo}}{\to} i$ was true will be present when $i_p \overset{\mathsf{vo}}{\to} i$.

Now that we have extended the definitions of executable and visible to include spontaneous pushes and spontaneous push order, we can add a "push-init" rule to the dynamic semantics.

This allows pushes to be spontaneously generated at any time. When a push is generated, a set of actions (all from the same thread) can be chosen to be made visible before the push (via spontaneous push-order):

$$\frac{\begin{array}{c}S \text{ is a set of actions} \\ \neg H(\mathsf{init}(i, *)) \\ \forall i' \in S.\, H(\mathsf{init}(i', p))\end{array}}{H \rightsquigarrow H, \mathsf{init}(i,p), \mathsf{is}(i, \mathsf{Push}), \mathsf{spec}(i,\ ()\ ),\ [\mathsf{spo}(i', i) \mid i' \in S]}\ (\textsc{Push-Init})$$

Specified push order also requires a new trace coherence rule, which we elide but present in Section A.

3.4.3 Spawning new threads

Since our model is fundamentally a language level model, there ought to be a way of creating new language threads. To do this, we add an expression form $\mathsf{fork}\, e$, which launches a new thread running the expression e. To implement it in the semantics, we additionally add a fork transaction $\mathsf{fork}\, e\, \mathsf{as}\, p$ which indicates that e is being forked off to run on thread p.

$$\begin{array}{lrcl} \text{expr's} & e & ::= & \cdots \mid \mathsf{fork}\, e \\ \text{transactions} & \delta & ::= & \cdots \mid \mathsf{fork}\, e\, \mathsf{as}\, p \end{array}$$

Typechecking fork is fairly straightforward. The expression to run must be well typed.

$$\frac{\Upsilon; \Phi; \Gamma \vdash e : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{fork}\, e : \mathsf{unit}}$$

A fork expression is stepped by returning unit and synchronizing on a fork transaction.

$$\frac{}{\mathsf{fork}\, e \overset{\mathsf{fork}\, e\, \mathsf{as}\, p}{\longmapsto} \mathsf{ret}\ ()}$$

Signatures are not affected by forking.

$$\frac{}{\Sigma \overset{\mathsf{fork}\, e\, \mathsf{as}\, p' @ p}{\longmapsto} \Sigma}$$

Since fork directly modifies execution states, they actually have an interesting rule for fork. When a fork occurs, the new thread is added to the execution state.

$$\frac{e \overset{\mathsf{fork}\, e''\, \mathsf{as}\, p'}{\longmapsto} e' \quad p' \text{ is fresh}}{(\xi \parallel p : e) \overset{\mathsf{fork}\, e''\, \mathsf{as}\, p' @ p}{\longmapsto} (\xi \parallel p : e' \parallel p' : e'')}$$

We then need to modify the primary execution state step rule to only handle non-fork cases:

$$\frac{e \overset{\delta}{\mapsto} e' \quad \delta \text{ is not of the form } \mathsf{fork}\, e''\, \mathsf{as}\, p'}{(\xi \parallel p : e) \overset{\delta @ p}{\longmapsto} (\xi \parallel p : e')}$$

All that's left is the store transition rule. This is fiddly, as we need away to synchronize events in the forking thread with events in the newly created thread. Ideally we would be able to use our normal tagging discipline for this, but it doesn't fit as well as we'd like it to: fork doesn't quite look enough like the other actions to have it make sense as an action, so tagging it would require making fork able to collect tags as well and there is no clear start of execution action that could be tagged, so we would have to add something.

So instead, forking has built-in pre and post edges. When a fork is initiated, we initiate a push on the forking thread with a visible pre edge and a nop on the forked thread with a visibility post edge. To make sure that the new thread is not executed until the old thread has finished any setup for it, we add a new $\mathsf{fxo}(i_1, i_2)$ event that will require that i_2 not be executed until i_1 has been.

- events $\theta \ ::= \ \cdots \mid \mathsf{fxo}(i, i)$
- *Fork-execution-order:* $i \overset{\mathsf{fxo}}{\to}_H i' \overset{\text{def}}{=} H(\mathsf{fxo}(i, i'))$.

$$\frac{\neg H(\mathsf{init}(i_1, *)) \quad \neg H(\mathsf{init}(i_2, *))}{H \overset{\mathsf{fork}\ e\ \mathsf{as}\ p_2 @ p_1}{\longmapsto} H, \mathsf{init}(i_1, p_1), \mathsf{is}(i_1, \mathsf{Push}), \mathsf{spec}(i_1, \texttt{()}), \mathsf{label}(\overset{\mathsf{vis}}{\lhd}, i_1), \\ \mathsf{init}(i_2, p_2), \mathsf{is}(i_2, \mathsf{Nop}), \mathsf{spec}(i_2, \texttt{()}), \mathsf{label}(\overset{\mathsf{exe}}{\rhd}, i_2), \\ \mathsf{fxo}(i_1, i_2)} \ (\textsc{Fork})$$

The one thing left is to extend the definition of executable to require that fork-execution-order predecessors have executed:

$$\begin{aligned} \mathsf{executable}_H(i) \overset{\text{def}}{=} & \neg \mathsf{exec}_H(i) \wedge (\forall i'.\, i' \overset{\mathsf{xo}}{\to}_H i \supset \mathsf{exec}_H(i')) \\ & \wedge (\forall i'.\, i' \overset{\mathsf{spo}}{\to}_H i \supset \mathsf{exec}_H(i')) \\ & \wedge (\forall i'.\, i' \overset{\mathsf{push}}{\Rightarrow}_H i \supset \exists i_p.\, H(\mathsf{is}(i_p, \mathsf{Push}) \wedge \mathsf{exec}_H(i_p)) \wedge i' \overset{\mathsf{spo}}{\to}_H i'') \\ & \wedge (\forall i'.\, i' \overset{\mathsf{fxo}}{\to}_H i \supset \mathsf{exec}_H(i')) \end{aligned}$$

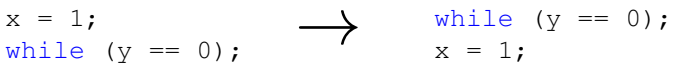
We also require a new trace coherence rule, which we elide but present in Section A.

3.4.4 Liveness side conditions

Consider the following program:

```
                          XEDGE(r, w);
x = 1;                    while (L(r, x) == 0);
while (y == 0);           L(w, y = 1);
```

The intended behavior here is that thread 1 writes to `x`, which thread 2 observes, allowing it to write to `y`, allowing thread 1 to finish. The problem with this is that under the semantics as discussed so far, the following compiler transformation is valid:

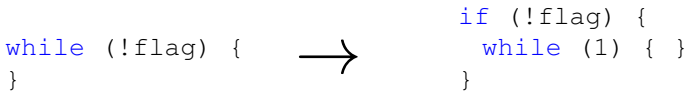


When executing the transformed version (on any actual hardware), the program will never terminate, since `x` will never be written to. This is allowed for two reasons. First, RMC is allowed to reorder actions when they are not constrained by an execution edge; thus, the write action can be indefinitely reordered past reads. Putting an execution edge in to rule this out, however, does not actually help us: while this would require to execute the write to `x`, nothing requires thread 2 to actually ever read from the write.

We resolve this problem by adding two "liveness" side conditions. We have always essentially depended on these properties implicitly (so that our programs do *something* instead of nothing), but they are spelled out in order to clearly rule out these transformations.

1. All executable actions will eventually be executed.
2. Writes will eventually become visible. That is, a thread will not indefinitely read from a write when there is another write coherence-after it.

The second side condition also rules out compiler transformations like the following (which compilers will perform on non-concurrent locations)



3.4.5 Sequentially consistent operations

Approach

As discussed in Section 2.3.2, we require that all operations on SC locations participate in a total order that is consistent with program order and is respected by reads (that is, reads read from the most recent write in the order). Additionally, they should have the appropriate message-passing behavior when used with weaker atomics and (the not-yet specified) non-atomics.

In the core calculus, instead of directly providing a fully realized SC atomic as described above, we make a minor extension that allows us to implement full SC atomics using it and existing features for constraint edges.

First, we add a notion of a SC operations to the calculus and we define built-in sequential consistency as trace-order over sequentially consistent operations. We then extend priority to include per-location built-in sequential consistency. Built-in sequential consistency is a total order over SC actions and it will be respected by reads (since it is reflected in priority, reading from the non-most-recent read would cause a coherence cycle).

While built-in sequential consistency agrees with trace order, it does not agree with program order as required. This can be easily rectified by tagging the sequentially consistent actions with $\overset{\text{exe}}{\triangleright}$. Because sequential consistent actions should have message-passing behavior when used

with weaker actions (and especially when used with non-atomic accesses), SC stores must be visibility-after prior actions, and so must be tagged with $\overset{\mathsf{vis}}{\lhd}$.

Details

We can now dive into the details of how sequentially consistent operations are formalized.

First, all memory access expressions and actions are extended with a tag saying whether they are regular concurrent accesses (c) or SC memory accesses (sc):

$$\begin{array}{llll} \text{expr's} & e & ::= & \cdots \\ & & \mid & \mathsf{R}_t[m] \mid \mathsf{W}_t[m,m] \\ & & \mid & \mathsf{RMW}_t[m, x{:}\tau.m, x{:}\tau.m] \\ & & \mid & \mathsf{RMWS}_t[\ell, v, m, m] \\ \text{actions} & \alpha & ::= & \cdots \\ & & \mid & \mathsf{R}_t[\ell] \mid \mathsf{W}_t[\ell, v] \mid \mathsf{RW}_t[\ell, v] \\ \text{action type} & t & ::= & \mathsf{c} \mid \mathsf{sc} \end{array}$$

Technically speaking, all of the rules that deal with these memory operations must now be updated to take into account the action type. In the interest of simplicity and legibility, we skip actually writing this all down. Usually it is irrelevant. Dynamic semantic transition rules preserve tags in the only sensible way.

Actions i and i' are ordered by built-in sequential consistency if both are tagged as sequentially-consistent actions and i is trace-order-before i'.

$$\mathsf{is\_type}_H(i,t) = H(\mathsf{is}(i, \mathsf{R}_t[\ell])) \vee H(\mathsf{is}(i, \mathsf{RW}_t[\ell,*])) \vee H(\mathsf{is}(i, \mathsf{W}_t[\ell,*]))$$

$$i \overset{\mathsf{bsc}}{\to}_H i' \overset{\text{def}}{=} \mathsf{is\_type}_H(i,\mathsf{sc}) \wedge \mathsf{is\_type}_H(i',\mathsf{sc}) \wedge i \overset{\mathsf{to}}{\to}_H i'$$

With $\overset{\mathsf{bsc}}{\to}$ defined, working it into coherence order is a simple matter of lifting it to $\overset{\mathsf{pri}}{\to}$.

$$\frac{i \overset{\mathsf{bsc}}{\to}_H i' \quad \mathsf{sameloc}_H(i,i')}{i \overset{\mathsf{pri}}{\to}_H i'} \text{ (PRI-SC)}$$

We can prove a theorem about built-in sequential consistency along the lines of the theorems from earlier:

Theorem 8 (Sequential consistency)**.** *Suppose H is consistent. Suppose further that S is a set of identifiers such that for all $i \in S$, $\mathsf{is\_type}_H(i,\mathsf{sc})$ and $H(\mathsf{label}(\overset{\mathsf{exe}}{\rhd}, i))$. Then there exists a sequentially consistent ordering for S.*

Proof: See Appendix B.1. □

RMC-C++ SC atomics

As discussed in Section 2.3.2, RMC-C++ does *not* provide direct access to a sc memory access type, but instead provides an `sc_atomic<T>` class that *only* provides sc operations with the proper execution and visibility constraints. If `store_sc` and `load_sc` are taken to be functions that

perform sc stores and loads, we can relate the behavior of the actual `sc_atomic<T>` class to the semantics given above by sketching a fictitious implementation in terms of sc actions and constraints:

```
template<typename T>
class sc_atomic {
private:
  rmc::atomic<T> val;

public:
  // ...
  void store(T v) {
    VEDGE(pre, write);
    XEDGE(write, post);
    L(write, val.store_sc(v));
  }
  T load() const {
    XEDGE(read, post);
    return L(read, val.load_sc());
  }
  // ...
};
```

3.4.6 Non-concurrent (plain) locations

Adding non-concurrent "plain" locations to RMC involves a bunch of minor changes.

Groundwork

First, we build on the "action types" introduced to implement SC atomics in Section 3.4.5 to add a tag for "plain" memory actions.

$$\begin{array}{rrcl} \text{action type} & t & ::= & \cdots \\ & & | & \mathsf{p} \end{array}$$

We then define the notion of "atomically-reads-from" as reads-from restricted to concurrent accesses, and then redefine visibility order in terms of atomically-reads-from.

- $\mathsf{concurrent}_H(i) = \mathsf{is\_type}_H(i, \mathsf{c}) \vee \mathsf{is\_type}_H(i, \mathsf{sc})$
- $i \overset{\mathsf{arf}}{\to}_H i' \overset{\text{def}}{=} H(\mathsf{rf}(i, i')) \wedge \mathsf{concurrent}_H(i) \wedge \mathsf{concurrent}_H(i')$
- $i \overset{\mathsf{vo}}{\to}_H i' \overset{\text{def}}{=} i \overset{\mathsf{vis}}{\Rightarrow}_H i' \vee i \overset{\mathsf{arf}}{\to}_H i' \vee i \overset{\pi\mathsf{o}}{\to}_H i' \vee i \overset{\mathsf{spo}}{\to}_H i'$

The purpose of all this is to exclude plain accesses from creating any sort of visibility between threads. This makes it easy to use visibility as the key notion in how we define data races below.

There is one remaining snag: the way that atomic reads are prevented from reading from a write that is program-order after it is that this would cause a cycle in priority and thus coherence. This does not work for plain reads, since reads-from on plain accesses does not contribute to visibility order. To solve this, we require, as a condition of reading from a location, that the write is prior to the read. For concurrent locations this condition is trivial, since the reads-from itself

establishes it. For plain locations, any earlier write on the same thread is clearly prior, and if there is an executed earlier write from another thread that isn't visible-to (and thus prior to) the, then there will be a data race, as discussed below.

$$\frac{\begin{array}{l} H(\mathsf{is}(i,\alpha)) \quad H(\mathsf{spec}(i,v)) \\ \alpha = \mathsf{R}[\ell] \vee \alpha = \mathsf{RW}[\ell, v'] \\ i_w \xrightarrow{\mathsf{pri}}_{H;\mathsf{rf}(i_w,i)} i \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H;\mathsf{rf}(i_w,i)}) \\ \mathsf{writes}_H(i_w,\ell,v) \quad \mathsf{exec}_H(i_w) \end{array}}{H \rightsquigarrow H, \mathsf{rf}(i_w, i)} \text{ (READ)}$$

Data races

The remaining thing is to specify more relaxed semantics for semantics for plain memory accesses. We do this in the most heavy handed way possible: by following C++'s lead and making data races between non-concurrent accesses be undefined behavior. We say that a state has a plain data race if there exist two plain conflicting accesses on different threads that are not ordered by visible-to.

We can specify all of this more precisely, though I don't think it sheds much additional light:

- $\mathsf{samethread}_H(i, i') = \exists p. H(\mathsf{init}(i, p))) \wedge H(\mathsf{init}(i', p))$
- $\mathsf{conflict}_H(i, i') = \exists l, v. \mathsf{sameloc}_H(i, i') \wedge (\mathsf{writes}_H(i, l, v) \vee \mathsf{writes}_H(i', l, v))$

$$\frac{\begin{array}{c} \mathsf{conflict}_H(i,i') \quad \neg\mathsf{samethread}_H(i,i') \quad \neg i \xrightarrow{\mathsf{vt}}_H i' \quad \neg i' \xrightarrow{\mathsf{vt}}_H i \\ \mathsf{is\_type}_H(i, \mathsf{p}) \vee \mathsf{is\_type}_H(i', \mathsf{p}) \end{array}}{(\Sigma, H, \xi)\ \mathsf{races}}$$

Then we say that if there exists *any* non-moot execution that contains a data race, the program is undefined:

$$\frac{s \mapsto^* s' \quad s'\ \mathsf{races} \quad \neg s'\ \mathsf{moot}}{s\ \mathsf{catches\text{-}fire}}$$

Discussion

This definition seems like it should work well. Using visible-to as the key mechanism to define data races means that in any program that does not catch fire, each plain read only has one choice of write to read from.

To see this, suppose that some plain data read i_r has two writes, i_w and i'_w that it can read from. This means that we have both $i_w \xrightarrow{\mathsf{pri}} i_r$ and $i'_w \xrightarrow{\mathsf{pri}} i_r$. Now, in order for two conflicting accesses not to race, they must be ordered by either program-order (because they are on the same thread) or by visible-to. Since priority includes both these orderings, i_w and i'_w must be related by priority or the program catches fire. Without loss of generality, suppose that $i_w \xrightarrow{\mathsf{pri}} i'_w$. Then, if $i_w \xrightarrow{\mathsf{rf}} i_r$, by CO-READ we have $i'_w \xrightarrow{\mathsf{co}} i_w$ and by CO-WRITE we have $i_w \xrightarrow{\mathsf{co}} i'_w$. Thus $i_w \xrightarrow{\mathsf{rf}} i_r$ induces a cycle in coherence order and is not valid, so i_r must read from i'_w.

What, if anything, is a data race?

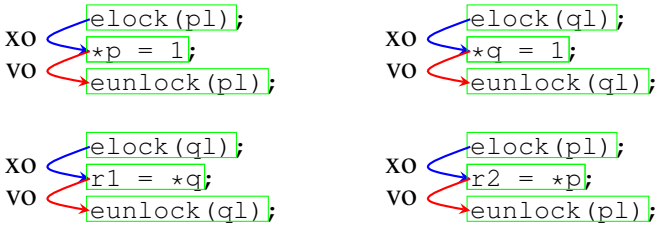
One snag in (more charitably, one subtlety with) this definition of data race is that it does not match the definition of data race used in the theorem (in Section 3.3.2) that data-race-free executions are sequentially consistent. That theorem uses a notion of *synchronized-before* that requires an execution post fence so that i will also be synchronized before program order successors of i':

$$i \overset{\mathsf{sb}}{\rightarrow}_H i' \overset{\text{def}}{=} \exists i_s.\, i \overset{\mathsf{vo}}{\rightarrow}^+_H i_s \overset{\mathsf{po}}{\rightarrow}^*_H i' \wedge H(\mathsf{label}(\overset{\mathsf{exe}}{\triangleright}, i_s))$$

This notion of data-race-freedom requires that all conflicting accesses on different threads be ordered by synchronized-before. Because the notion of data-race-freedom using synchronized-before is strictly stronger than the version defined using visible-to, we will refer to it as being *strong data-race-freedom* (and of executions satisfying it as being *strongly data-race-free*. During this discussion we will refer to the visible-to version as *strong data-race-freedom* (and of executions as *weakly data-race-free*).

Why do we need a different definition of data-race for our sequentially consistent fragment and for the definition of non-atomic locations? It is because the goals of ruling out non-SC behaviors and compiling plain locations have fundamentally different constraints. Weak data-race-freedom is not strong enough to guarantee sequential consistency and strong data-race-freedom would rule out many perfectly good programs if used to trigger undefined behavior. A simple example program will serve to demonstrate this.

First, consider the following program:



Here, we take `elock` and `eunlock` to be mutex lock and unlock functions that lack the usual mutex pre- and post- constraints and thus must explicitly have edges drawn to use them. We then use these explicit locks to protect the access to shared locations in a standard store-buffering example.

This program is weakly data-race-free but not strongly data-race-free. Weakly because all of the accesses to `*p` and `*q` will be ordered by visible-to (assuming that `elock` reads-from the previous `eunlock`) and not strongly because there are no execution post edges that are needed for synchronized before.

And although the program is weakly data-race-free, it *does* allow the non-sequentially-consistent behavior of `r1 == 0`, `r2 == 0`. The code of each thread consists of two independent blocks of code (a write block and a read block) that have no constraints between them. Nothing prevents an entire read block from being reordered and executed before the entirety of the write block. Thus, weak data-race-freedom is insufficient to ensure sequential consistency.

Additionally, although the program is not strongly data-race-free, it ought to be perfectly well defined. In any given execution of the program, the reads are fully determined by the order

in which each lock is acquired: if thread 1 locks `ql` first then we will have `r1 == 0`; if thread 2 acquires it first, we'll get `r1 == 1`. Furthermore, the accesses to each shared variable can't happen "at the same time"; even if they held multi-word objects that needed to be stored and loaded in multiple non-atomic steps, because the locks protect them, still only 0 or 1 may be read.

While this might not be the most compelling of examples (as locks really ought to have pre and post edges) for why it is necessary to use weak data-race-freedom to define the semantics of plain accesses, it is critical if we want to allow essentially any use of fine-grained edges in combination with plain actions. We do this heavily in our RCU-protected linked lists, discussed in Section 5.7.

3.4.7 Allocations

Our system as presented so far lacks any way to allocate new memory locations, which is a large limitation.

$$\begin{array}{lrcl} \text{expr's} & e & ::= & \cdots \mid \mathsf{alloc}\ m{:}\tau \\ \text{transactions} & \delta & ::= & \cdots \mid \mathsf{alloc}\ v{:}\tau \mathsf{as}\ \ell \end{array}$$

The typing rule and the compatibility rule for the dynamics are pretty straightforward:

$$\frac{\Upsilon; \Phi; \Gamma \vdash m : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{alloc}\ m{:}\tau : \tau\ \mathsf{ref}} \qquad \frac{m \mapsto m'}{\mathsf{alloc}\ m{:}\tau \overset{\emptyset}{\mapsto} \mathsf{alloc}\ m'{:}\tau}$$

The "interesting" evaluation rule is also pretty straightforward. It just synchronizes on the new transaction:

$$\frac{}{\mathsf{alloc}\ v{:}\tau \overset{\mathsf{alloc}\ v:\tau \mathsf{as}\ \ell}{\longmapsto} \mathsf{ret}\ \ell}$$

Signature evaluation typechecks v and makes sure that ℓ is distinct.

$$\frac{\Upsilon; \Phi; \epsilon \vdash v : \tau \quad \ell \notin \mathsf{Dom}(\Phi)}{(\Upsilon, \Phi) \overset{\mathsf{alloc}\ v:\tau \mathsf{as}\ \ell @ p}{\longmapsto} (\Upsilon, (\Phi, \ell{:}\tau))}$$

The interesting work is all on the history side, and it is a doozy. The main difficulty in allocation is that our static semantics require that every location in the signature have an executed initialization write that is visibility-prior to an executed push that is trace order before all reads from the location. While that rule in particular could be tweaked, it is an important safety property that there is at least one write prior to every read.

In order to accomplish this, we need to make sure that as soon as the label is introduced into the context, a write and a push have been executed. This, unfortunately, does not at all fit nicely with the standard RMC model of transactions being initiated by the thread semantics and then executing later, at their leisure. So instead we add a synchronous store transition that is not at all in the spirit of RMC: the ALLOC rule both initiates and *executes* an initial write for the newly allocated location and a push to make it visible. In order to ensure that these actions execute after any execution-order prior actions, it actually stalls until the actions that it introduces would be executable.

$$\frac{\begin{array}{l}\neg H(\mathsf{init}(i_w, *)) \\ \neg H(\mathsf{init}(i_p, *)) \\ H' = H, \mathsf{init}(i_w, p), \mathsf{is}(i_w, \mathsf{W_c}[\ell, v]), \mathsf{spec}(i_w, \texttt{()}), \\ \qquad \mathsf{init}(i_w, p), \mathsf{is}(i_w, \mathsf{Push}), \mathsf{spec}(i_p, \texttt{()}), \\ \qquad \mathsf{spo}(i_w, i_p) \\ \\ \mathsf{executable}_{H'}(i_w) \qquad \mathsf{executable}_{H';\mathsf{exec}(i_w)}(i_p) \\ \mathsf{acyclic}(\overset{\mathsf{co}}{\rightarrow}_{H',\mathsf{exec}(i_w),\mathsf{exec}(i_p)})\end{array}}{H \overset{\mathsf{alloc}\ v:\tau\,\mathsf{as}\,\ell@p}{\longmapsto} H', \mathsf{exec}(i_w), \mathsf{exec}(i_p)}\ (\textsc{Alloc})$$

Perhaps unusually, there is no corresponding trace coherence rule. There is no event that records an allocation in the history, so we just need to ensure that the history produced by the allocation rule obeys the coherence rules for all the events it creates.

OK, but seriously, that push is awful

In a C++-style setting, actually executing a push whenever a a new location (which may well be a stack location!) is allocated is certainly a nonstarter. There are a couple approaches that we could take to repair this. The most straightforward is to relax both the allocation rule to not include a push and the static semantics. This will likely compromise some soundness properties, but since we have catch-fire semantics for data-races in RMC-C++ we arguably don't care. There is another approach that allows us to reuse these semantics but is also a hack: to model a C++ allocation as an RMC allocation followed by a non-atomic write to the location. Then, anything that races with the initialization will invoke catch-fire behavior and the push can be "optimized out".

Chapter 4

Compiling RMC

In order to properly evaluate whether programmer specified constraints are a practical approach, we need to be able to use them in a real programming language with a practical compiler. To this end, I have built `rmc-compiler`[43], an extension to LLVM which accepts specifications of ordering constraints and compiles them appropriately. With appropriate support in language frontends, this allows any language with an LLVM backend to be extended with RMC support. Languages with some sort of macro facility and an easy way to call C functions are fairly easy to support: C, C++, and Rust have been extended with RMC style atomic operations using macro libraries that expand to the specifications expected by `rmc-compiler`.

4.1 General approach

A traditional way to describe how to compile language concurrency constructs—as demonstrated in [7]—is to give direct mappings from language constructs to sequences of assembly instructions. This approach works well for the C++11 model, in which the behavior of an atomic operation is primarily determined by properties of the operation (its memory order, in particular). In RMC, however, this is not the case. The permitted behavior of atomic operations (and thus what code must be generated to implement them) is primarily determined by the edges between actions. Our descriptions of how to compile RMC constructs to different architectures, then, focus on edges, and generally take the form of determining what sort of synchronization code needs to be inserted *between* two labeled actions with an edge between them.

In discussion of the compiler, we use "action" in a slightly different way than we do in describing the semantics—in the terminology of the compiler, an action is any labeled expression or statement that can have constraints connected to it. Actions will typically be a single read or write, but can contain arbitrary code.

4.2 x86

While it falls short of sequential consistency, x86's memory model [42] is a pleasure to deal with. Unlike ARM and POWER, it is actually productive to model x86 as having a main memory, albeit one supplemented by per-CPU FIFO store buffers of writes that have not yet been propagated to

it. Although real x86 processors execute things aggressively out of order, they do not do so in any observable way, and so we are free to model x86 as executing everything in order. And while x86 processors can have complicated memory systems, the only observable behavior of them is the aforementioned store buffers.

Because no reordering of instructions is observable on x86, nothing needs to be done to ensure execution order on the CPU side. If two memory operations appear in a certain order in the assembly, they will execute in that order.

Because the store buffers are FIFO, writes become visible to other processors in the order they were executed. This means that if a write is visible to some other CPU, all writes program order before that write (in the assembly) are visible as well. Thus, like for execution, nothing CPU-specific needs to be done to ensure visibility order.

The one catch in the above, however, is that in order for x86 to preserve execution and visibility order, the relevant actions must still occur in program order in the generated x86 code. That is, while nothing needs to be done to keep the x86 *processor* from reordering constrained actions, we do still need to keep the compiler from doing so. So while we do not insert any hardware barriers for edges on x86, we do insert a "compiler barrier" into the LLVM intermediate-representation between the source and destination of each edge. The compiler barrier does not generate any code but does inhibit compiler transformations from moving memory accesses across it.

Even on x86, though, push is not free. A push requires everything that is visible to it to become globally visible when it executes. On x86, this is exactly what `MFENCE` does: it drains the store buffer, requiring all previous writes on the CPU to become globally visible (since x86's only inter-CPU visibility is global, any visible writes from other CPUs are *already* globally visible.

4.3 ARMv7 and POWER

Life is not so simple on ARM and POWER, however. [1] POWER has a substantially weaker memory model [40] than x86 that incorporates both a very weak memory subsystem in which writes can propagate to different threads in different orders (that do not correspond to the order they executed) *and* visible reordering of instructions and speculation. For most of our purposes, ARM's model [4] is quite similar to POWER, though writes may not propagate to other threads in different orders.

Compiling visibility edges is still fairly straightforward. POWER provides an `lwsync` ("lightweight sync") instruction that does essentially what we need: if a CPU A executes an `lwsync`, no write after the `lwsync` may be propagated to some other CPU B unless *all* of the writes propagated to CPU A (including its own) before the `lwsync`—the barrier's "Group A writes", in the terminology of POWER/ARM—have also been propagated to CPU B. That is, all writes before (including those observed from other CPUs) the `lwsync` must be visible to another thread before the writes after it. Then, executing an `lwsync` between the source and destination of a visibility edge is sufficient to guarantee visibility order. The strictly stronger `sync` instruction on POWER is also sufficient. ARM does not have an equivalent to POWER's `lwsync`, and

[1] We begin by restricting our attention to ARMv7. ARMv8 adds a number of new memory ordering constructs and is discussed in Section 4.5

so—in the general case—we must use the stronger `dmb`, which behaves like `sync`. ARM does, however, have the `dmb st` instruction, which requires that all stores on CPU A before the `dmb st` become visible to other CPUs before all stores after the barrier, but imposes no ordering on loads. This is sufficient for implementing visibility edges between simple stores.

To implement pushes, we turn to this stronger barrier, `sync`. The behavior of `sync` (and `dmb`) is fairly straightforward: the `sync` does not complete and no later memory operations can execute until all writes propagated to the CPU before the `sync` (the "Group A writes") have propagated to all other CPUs. This is essentially exactly what is needed to implement a push.

While compiling visibility edges and pushes is fairly straightforward and does not leave us with many options, compiling execution edges presents us with many choices to make. ARM and POWER have a number of features that can restrict the order in which instructions may be executed:

- All memory operations prior to a `sync`/`lwsync` will execute before all operations after it.
- An `isync` instruction may not execute until all prior branch targets are resolved; that is, until any loads that branches are dependent on are executed. Memory operations cannot execute until all prior `isync` instructions are executed.
- A write may not execute until all prior branch targets are resolved; that is, until any loads that the control is dependent on are executed.
- A memory operation can not execute until all reads that the address or data of the operation depend on have executed.

All of this gives a compiler for RMC a bewildering array of options to take advantage of when compiling execution edges. First, existing data and control dependencies in the program may already enforce the execution order we desire, making it unnecessary to emit any additional code at all. When there is an existing control dependency, but the constraint is from a read to a read, we can insert an `isync` after the branch to keep the order. When dependencies do not exist, it is often possible to introduce bogus ones: a bogus branch can easily be added after a read and, somewhat more disturbingly, the result of a read may be xor'd with itself (to produce zero) and then added to an address calculation! And, of course, we can always use the regular barriers.

This gives us a toolbox of methods with different characteristics. The barriers, `sync` and `lwsync`, enforce execution order in a many-to-many way: all prior operations are ordered before all later ones. Using control dependency is one-to-many: a single read is executed before either all writes after a branch or all operations after a branch and an `isync`. Using data dependencies is one-to-one: the dependee must execute before the depender. As C++ struggles with finding a variation of the "consume" memory order that compilers are capable of implementing by using existing data dependencies, we feel that the natural way in which we can take advantage of existing data dependencies to implement execution edges is one of our great strengths.

4.4 Optimization

4.4.1 General Approach

The huge amount of flexibility in compiling RMC edges poses both a challenge and an opportunity for optimization. As a basic example, consider compiling the following program for ARM:

```
VEDGE(wa, wc);
VEDGE(wb, wd);
L(wa, a = 1);
L(wb, b = 2);
L(wc, c = 3);
L(wd, d = 4);
```

This code has four writes and two edges that overlap with each other. According to the compilation strategy presented above, to compile this on ARM we need to place a `dmb` somewhere between `wa` and `wc` and another between `wb` and `wd`. A naive implementation that always inserts a `dmb` immediately before the destination of a visibility edge would insert `dmb`s before `wc` and `wd`. A more clever implementation might insert `dmb`s greedily but know how to take advantages of ones already existing—then, after inserting one before `wc`, it would see that the second visibility edge has been cut as well, and not insert a second `dmb`. However, like most greedy algorithms, this is fragile; processing edges in a different order may lead to a worse solution. A better implementation would be able to search for places where we can get more "bang for our buck" in terms of inserting barriers.

Things get even more interesting when control flow is in the mix. Consider these programs:

```
VEDGE(wa, wb);
L(wa, a = 1);
if (something) {
  L(wb, b = 2);
  // other stuff
}
```

```
VEDGE(wa, wb);
L(wa, a = 1);
while (something) {
  L(wb, b = 2);
  // other stuff
}
```

In both of them, the destination of the edge is executed conditionally. In the first, it is probably better to insert a barrier *inside* the conditional, to avoid needing to execute it. The second, with a loop, is more complicated; which is better depends on how often the loop is executed, but a good heuristic is probably that the barrier should be inserted outside of the loop. [2] Furthermore, consider a program such as:

[2] Or, though RMC will not do this itself, that the loop should have separate branches for entering the loop and staying in it.

```
VEDGE(wa, wb);
L(wa, a = 1);
if (something) {
  // stuff
} else {
  // other stuff
}
L(wb, b = 2);
```

Broadly speaking, there are three choices for where to insert barriers here: immediately after `wa`, immediately before `wb`, or inside *both* branches of the conditional. We need to execute a barrier along every path from `wa` to `wb`, but it need not be the *same* barrier along each path.

A CFG-centric approach

Because much of the trickiness in compiling RMC is in the handling of control flow, we use a control flow graph (CFG)-centric approach to compilation in which each labeled action is placed in a basic block by itself. [3] We can then view paths between actions as paths between nodes in the control flow graph and we can limit our insertions of barriers to the beginning and end of basic blocks. To handle edges between invocations of a function, we extend the CFG to contain edges from all exits of the function to the entrance of the function, in order to model paths into future invocations of the function.

4.4.2 Analysis and preprocessing

In order to take advantage of the wide array of compilation options discussed above, we need to properly categorize our actions. We inspect the bodies of actions in order to assign one of five categories to them: simple read (if they contain a single load instruction), simple write (if they contain a single store instruction), RMW (if they contain a single read-modify-write instruction), no-op (if they contain no memory operations and no function calls), and complex (for everything else).

After the analysis phase, we do some preprocessing steps on the graph of actions to make it more amenable to our CFG-centric SMT compilation strategy.

Pre and post edges

RMC's "pre" and "post" edges establish constraints with *all* program order predecessors or successors. To handle these edges, for each labeled action in the program we generate pre- and post- "dummy" actions. These are actions located immediately before and after the labeled action in the control flow graph; they contain no instructions but are treated as complex actions.

[3] Or in a collection of basic blocks, if the action has internal control flow. We will mostly ignore this subtlety in our discussion, though our compiler doesn't have this luxury.

Pushes and push edges

While the RMC formalism treats pushes as the primitive and push edges as a convenient abbreviation, our compilation strategy takes the reverse view and operates in terms of compiling push edges directly. What we would *like* to do to convert explicit pushes into push edges is to find every triple of actions such that $a \xrightarrow{\text{vo}} \text{push} \xrightarrow{\text{xo}} b$ and generate a push edge from a to b. Unfortunately, pre and post edges make it impossible to find all such actions, short of sophisticated global analyses. So instead we convert explicit push actions are into push edges in a mildly hokey manner: using pre and post edges, we convert every explicit push into a push edge from all program-order predecessors to all program-order successors. The somewhat unfortunate outcome of this is that, from the compiler's perspective, it renders totally moot the elegant interaction of pushes with visibility and execution edges—all explicit pushes will result in a full fence at exactly where it is written, regardless of visibility and execution edges.

The action graph

To drive the compilation, we construct an "action graph" with actions as the nodes and a set of edges for each of the three types of constraint edges. Because specified edges in RMC are transitive, an edge can have meaning even when at first glance it appears meaningless (for example, an execution edge between two no-ops!). We eliminate this subtlety by explicitly taking the transitive closure of the action graph. [4] Specified visibility order implies execution order, and this must be taken into account while computing this transitive closure; we handle this by, prior to computing the transitive closure, explicitly adding execution edges to our graph for each visibility edge. Specified push order also implies both visibility and execution order, but because we (as discussed below) never prune push edges from our action graph, we don't explicitly include push order in our other orders.

Once this graph has been constructed, we are able to consider each edge in isolation without regard to its transitive implications or it implying weaker edges. This is useful because it allows us to consider each constraint in isolation and to ignore edges meaningful only for their transitive implications.

- Execution edges with a simple write as the source have no meaningful effect. There are two components to this, because execution edges have implications in both the memory system model and the execution model. On the execution model side, the only implication of a write having executed is that reads *may* read from it. However, absent other constraints, there's no observational difference between a write having not executed and a write having executed but reads not reading from it yet.

  On the memory system side, execution edges from a write can't meaningfully participate in visible-to (once the transitivity of specified edges is ignored). They are unable to connect to visibility via reads-from (because the source is not a read!).

  An execution edge from a write *can* interact with push order, but as it turns out not in a meaningful way. An execution edge from a write i_w to i' means that i' must be push-ordered after any push that executes before i_w does. A write i_w executing after a push

[4] Using everyone's favorite $O(n^3)$ algorithm, the Floyd-Warshall algorithm

means that i_w must be coherence-after all writes that are visible to the push—but this is indistinguishable from i_w executing before the push and just happening to come coherence-after those writes. Because there is no way, absent other constraints, to tell for sure that i_w executed after a push, a condition that applies only when i_w executed after a push is not actually meaningful.

Thus, requiring a write executes before another action accomplishes nothing.

- Visibility edges to simple reads have no effect beyond that of the execution edge that it implies. This is because visibility is only important when it can participate in a visible-to relationship. Once transitivity of specified edges is ignored, the only way a visibility edge can meaningfully participate in a visible-to relationship is if it is followed by push-order, reads-from, or spontaneous-push-order in the visible-to relation. A visibility edge ending in a read can not do either of the first two.

  Spontaneous-push-order arises from specified push edges, and is slightly more subtle: if we have a specified visibility edge from i to a simple read i_r and a specified push edge from i_r to i', this implies that i must be visible before a push that executes before i'. Since an equivalent effect could be obtained by having a specified push edge from i to i' directly, this is *also* a form of transitivity, and not any inherent meaning of visibility edges to writes. The compilation of push edges already needs to be robust to unknown actions being visible before the push, so this doesn't add any extra snags.
- Visibility and execution edges where one of the actions is a no-op have no effect at all. This includes actions that were pushes that got converted to push edges.
- Push edges can never be removed. Because pushes interact with visibility and execution order in an important way and because (as discussed above) it is impossible to find all actions that are visibility ordered before a push, it is never safe to delete a push edge.

Binding sites

One snag in the above discussion of the action graph is that it does not take into account that edges declared using `VEDGE_HERE` and friends have a binding site associated with them. Like actions themselves, binding sites are merely basic blocks. Additionally, it is possible (though probably not particularly useful), to actually have edges with the same source and destination but different binding sites. It is fairly simple, then, to generalize the action graph into a multigraph[5] in which edges can be labeled with binding sites.

The only subtlety here, then, is how to take the transitive closure of the action graph with binding sites. The trickiness arises when we need to determine how to create a new transitive edge from two edges that have different binding sites. The answer winds up being fairly simple. In RMC, it is always sound to add additional constraints to the program. Because the scope of a scoped constraint is defined as all the basic blocks dominated by the binding site, it is sound to "widen" the scope of the constraint by moving the binding site to a block that dominates the original binding site (or to eliminate the binding site entirely, making the edge bound outside

[5]Though I guess it technically already was, because of the multiple types of actions. Though that can more easily be viewed as separate graphs.

the function). Then, in order to join two edges with different binding site, we find the nearest common dominator of the two binding sites and use this as the binding site for the joined edge (or have no binding site if either of the original edges had none). Using a common dominator as the binding site is sound because it extends the scope of both of the constraints being joined. Using the *nearest* common dominator allows us to avoid growing the scope more than we have to.

Downside of transitive closure

While taking the transitive closure of the action graph is quite simple and has a number of important benefits, it is not without drawbacks. In the semantics of RMC, transitivity of actions is a *dynamic* property, since only actions that are actually executed can participate. The transitive closure of the action graph that we compute, on the other hand, is static. Consider the program:

```
VEDGE_HERE(a, b);
VEDGE_HERE(b, c);
bool cond = L(a, val);
if (cond) {
   L(b, noop());
}
L(c, stuff());
```

By the RMC semantics, if `cond` is false, the `b` action is never executed and so doesn't exist in any meaningful sense. If `cond` is true and it does execute, we have $a \xrightarrow{\text{vo}} b \xrightarrow{\text{vo}} c$ and thus $a \xrightarrow{\text{vo}^*} c$. But if it doesn't execute, `b` is not present to make the transitive connection and we do *not* have $a \xrightarrow{\text{vo}^*} c$. Since we only require $a \xrightarrow{\text{vo}^*} c$ when `cond` is true, we would like to only insert an `lwsync` when the conditional is taken.

Unfortunately our method of computing the transitive closure of the action graph works against us here. We will see that there are edges from `a` to `b` and from `b` to `c` and create an edge from `a` to `c`. The original two edges will then be deleted because they involve a no-op, and a barrier will be inserted unconditionally between `a` and `b`.

4.4.3 Compilation Using SMT

We model the problem of enforcing the constraints as an SMT problem and use the Z3 SMT solver [20] to compute the optimal placement of barriers and use of dependencies (according to our metrics). The representation we use was inspired by the integer-linear-programming representation of graph multi-cut [17]—we don't go into detail about modeling our problem as graph multi-cut, since it is not actually particularly more illuminating than the direct SMT representation is and it does not scale up to using dependencies. This origin survives in our use of the word "cut" to mean satisfying a constraint edge.

Using integer linear programming (ILP) to optimize the placement of barriers is a common technique. Bouajjani et al. [14] and Alglave et al. [1] both use ILP to calculate where to insert barriers to recover sequential consistency as part of tools for that purpose. In a recent paper, Bender et al. [8] use ILP to place barriers in order to compile what is essentially a simplified

version of RMC (with only one sort of edge, most akin to RMC's derived "push edges"). We believe we are the first to extend this technique to handle the use of dependencies for ordering. [6]

Compilation proceeds a function at a time. Given the set of labeled actions and constraint edges and the control flow graph for a function, we produce an SMT problem with solutions that indicate where to insert barriers and where to take advantage of (or insert new) dependencies. The SMT problem that we generate is *mostly* just a SAT problem, except that integers are used to compute a cost function, which is then minimized. Z3 now has built in support for minimizing a quantity, but even without that, the cost can be minimized by adding a constraint that bounds it, and then binary searching on that bound.

We present two versions of this problem. As an introduction, we first present a complete system that always uses barriers, even when compiling execution edges. We then discuss how to generalize it to use control and data dependencies. We will primarily use the terminology of POWER.

Barrier-only implementation

The rules for encoding the compilation of visibility edges as an SMT problem are reasonably straightforward:

$$\begin{aligned} & \bigwedge_{s \xrightarrow{\mathsf{vo}} t} \mathsf{vcut}(s,t) \\ \mathsf{vcut}(s,t) &= \bigwedge_{p \in \mathsf{paths}(s,t)} \mathsf{vcut\_path}(p) \\ \mathsf{vcut\_path}(p) &= \bigvee_{e \in p} \overline{\mathsf{lwsync}}(e) \vee \overline{\mathsf{sync}}(e) \end{aligned}$$

We write $\mathsf{foo}(x)$ to mean a variable in the SMT problem that is given a definition by our equations and $\overline{\mathsf{foo}}(x)$ to mean an "output" variable whose value will be used to drive compilation. Later in this chapter, we will use $foo(x)$ to mean an "input" variable that is not a true SMT variable at all, but a constant set by the compiler based on some analysis of the program.

Here, the assignments to the $\overline{\mathsf{lwsync}}$ and $\overline{\mathsf{sync}}$ variables produced by the SMT solver are used by the compiler to determine where to insert barriers. We write $s \xrightarrow{\mathsf{vo}} t$ to quantify over visibility edges from s to t and take $\mathsf{paths}(s,t)$ to mean all of the simple paths from s to t. [7] Knowing that, these rules state that (1) every visibility edge must be cut, (2) that to cut a visibility edge, each path between the source and sink must be cut, and (3) that to have a visibility cut on a path means deciding to insert `sync` or `lwsync` at one of the edges along the path.

Since in the version we are presenting now, we only use barriers to enforce execution order, the condition for an execution edge is the same as that for a visibility one (writing $s \xrightarrow{\mathsf{xo}} t$ to

[6] The original barrier-only version of this algorithm for RMC was developed by Salil Joshi as an ILP problem, and I extended it to use dependencies, ruining the "linear" part and requiring an upgrade to an SMT solver.

[7] By "simple path" we mean a path that does not include any repeated vertexes *except* that the first and last vertex may be the same. The last condition is important to allow us to talk about paths from a node to itself.

quantify over execution edges):

$$\bigwedge_{s \xrightarrow{\mathsf{xo}} t} \mathsf{vcut}(s,t)$$

The rules for compiling push edges are straightforward: they are essentially the same as for visibility, except only heavyweight syncs are sufficient to cut an edge (writing $s \xrightarrow{\mathsf{pu}} t$ to quantify over push edges):

$$\begin{aligned} & \bigwedge_{s \xrightarrow{\mathsf{vo}} t} \mathsf{pcut}(s,t) \\ \mathsf{pcut}(s,t) &= \bigwedge_{p \in \mathsf{paths}(s,t)} \mathsf{pcut\_path}(p) \\ \mathsf{pcut\_path}(p) &= \bigvee_{e \in p} \overline{\mathsf{sync}}(e) \end{aligned}$$

All of the rules shown so far allow to find *a* set of places to insert barriers, but we could have done that already without much trouble. We want to be able to *optimize* the placement. This is done by minimizing the following quantity:

$$\sum_{e \in E} \overline{\mathsf{lwsync}}(e) w(e) cost_{\mathsf{lwsync}} + \overline{\mathsf{sync}}(e) w(e) cost_{\mathsf{sync}}$$

Here, we treat the boolean variable representing barrier insertions as 1 if they are true and 0 if false. The $w(e)$ terms represent the "cost" of an edge—these are precomputed based on how many control flow paths travel through the edge and whether it is inside of loops. The $cost_{\mathsf{lwsync}}$ and $cost_{\mathsf{sync}}$ terms are weights representing the costs of the `lwsync` and `sync` instructions, and should be based on their relative costs.

Differences on ARM

As mentioned earlier, ARM does not have an equivalent to `lwsync`, but does have `dmb st`, which suffices to ensure visibility ordering when the source is a simple (not an RMW) write. While `dmb st` only ensures ordering from writes to writes, we don't actually need to worry about what sort of action the destination is, since visibility edges only carry force when they are to writes.

Using this requires just simple modifications to the definition of vcut_path (and to the cost function, which we elide):

$$\begin{aligned} \mathsf{vcut\_path}(p) &= \bigvee_{e \in p} \mathsf{dmb}(e) \vee \mathsf{do\_dmbst}(p,e) \\ \mathsf{do\_dmbst}(p,e) &= \overline{\mathsf{dmbst}}(e) \wedge is\_simple\_write(s) \\ & \quad \text{(where } s = \mathsf{head}(p)\text{)} \end{aligned}$$

Here, is_simple_write(t) is also an input that is true if t is an action containing only writes.

We allow a path to be visibility cut by a `dmb` along the path or—if the source of the edge is a simple write—by a `dmb st`.

Dependency trickiness

The one major subtlety that needs to be handled when using dependencies to enforce execution ordering is that ordering must be established with *all* subsequent occurrences of the destination. Consider the following code:

```
void f(rmc::atomic<int> *p, rmc::atomic<int> *q, bool b) {
  XEDGE(ra, wb);
  int i = L(ra, *p);
  if (b) return;
  if (i == 0) {
    L(wb, *q = 1);
  }
}
```

In this code, we have an execution edge from `ra` to `wb`. We also have a control dependency from `ra` to `wb`, which we may want to use to enforce this ordering. There is a catch, however—while the execution of `wb` is always control dependent on the result of the `ra` execution from the current invocation of the function, it is *not* necessarily control dependent on executions of `ra` from previous invocations of the function (which may have exited after the conditional on `b`).

The takeaway here is that we must be careful to ensure that the ordering applies to all future actions, not just the closest. Just because an action is dependent on a load does not mean it is necessarily dependent on all prior invocations of the load. Our solution to this is, when using a dependency to order some actions A and B, to additionally require that A be execution ordered with subsequent invocations of itself. If we are are using a control dependency to order A and B, we can get a little weaker—it suffices for future executions of A to be control dependent on A, even if that would not be enough to ensure execution order on its own.

Supporting dependencies

With this in mind, we can now give the constraints that we use for handling execution order. They are considerably more hairy than those just using barriers. First, the "top-level rules":

$$
\begin{aligned}
& \bigwedge_{s \xrightarrow{\mathsf{xo}} t} \mathsf{xcut}(s,t) \\
\mathsf{xcut}(s,t) \;=\; & \bigwedge_{p \in \mathsf{paths}(s,t)} \mathsf{xcut\_path}(p) \\
\mathsf{xcut\_path}(p) \;=\; & \mathsf{vcut\_path}(p) \vee \\
& (\mathsf{ctrlcut\_path}(p) \wedge (\mathsf{ctrlcut}(s,s) \vee \mathsf{xcut}(s,s))) \vee \\
& (\mathsf{datacut\_path}(p) \wedge \mathsf{xcut}(s,s)) \\
& (\text{where } s = \mathsf{head}(p))
\end{aligned}
$$

As discussed above, execution order edges can be cut by barriers, like with visibility, and also by control and data dependencies, given that ordering can be established with future executions of the source action.

The control dependency related constraints are more involved:

$$\begin{aligned}
\mathsf{ctrlcut_path}(p) &= (is_simple_write(t) \wedge \mathsf{ctrl_path}(p)) \vee \mathsf{ctrlisync_path}(p) \\
&\quad \text{(where } t = \mathsf{tail}(p)\text{)} \\
\mathsf{ctrlcut}(s,t) &= \bigwedge_{p \in \mathsf{paths}(s,t)} \mathsf{ctrl_path}(p) \\
\\
\mathsf{ctrl}(s,e) &= can_ctrl(s,e) \wedge \overline{\mathsf{use_ctrl}}(s,e) \\
\mathsf{ctrl_path}(p) &= \bigvee_{e \in p} \mathsf{ctrl}(s,v) \\
&\quad \text{(where } s = \mathsf{head}(p)\text{)} \\
\mathsf{ctrlisync_path}(p) &= \bigvee_{e::p' \in p} \mathsf{ctrl}(s,e) \wedge \mathsf{isync_path}(p') \\
&\quad \text{(where } s = \mathsf{head}(p)\text{)} \\
\mathsf{isync_path}(p) &= \bigvee_{e \in p} \overline{\mathsf{isync}}(e)
\end{aligned}$$

In this, $can\_ctrl(s,e)$ is an input to the problem and is true if there is—or it would be possible to add—a branch along the edge e that is dependent on the value read in s. Then, $\overline{\mathsf{use\_ctrl}}(s,e)$ is an output indicating we are depending on that control dependency and so it must be either preserved or added. In the notation $e :: p' \in p$, we intend for e to represent an edge in p and p' to represent the rest of the path after e, starting with e's destination.

There are a lot of moving parts here, but the main idea is simple: an execution edge can be cut along a path by a control dependency followed by an `isync`. If the target of the constraint is a write, then we don't need the `isync`. The key equation defining $\mathsf{ctrlcut\_path}(p)$ expresses this. Then, $\mathsf{ctrl\_path}(p)$ states that there is a control dependency along p while $\mathsf{ctrlisync\_path}(p)$ says that there is a control dependency followed by an `isync` later in the path; $\mathsf{ctrlcut}(s,t)$ asserts that there is a control dependency on s along *every* path from s to t.

Data dependencies manage to be a little bit simpler to handle:

$$\begin{aligned}
\mathsf{datacut_path}(p) &= \bigvee_{(_,t)::p' \in p} \mathsf{data}(s,t,p) \wedge (\mathsf{ctrlcut_path}(p') \vee \mathsf{datacut_path}(p')) \\
&\quad \text{(where } s = \mathsf{head}(p)\text{)} \\
\mathsf{data}(s,t,p) &= can_data(s,t,p) \wedge \overline{\mathsf{use_data}}(s,t,p)
\end{aligned}$$

The idea here is straightforward: a constraint is cut by data dependencies if there is a chain of data deps—possibly concluded by a control dep—from the source to the sink. (Note that $(\mathsf{ctrlcut\_path}(p') \vee \mathsf{datacut\_path}(p'))$ is vacuously true if the path is empty.)

Here, $can\_data(s,v,p)$ is an input to the problem and is true if there is a data dependency from s to v, following the path p (it could also be extended to mean that a dependency could be *added*, but the compiler does not currently do that) while $\overline{\mathsf{use\_data}}(s,v,p)$ indicates that we are depending on the dependency and the compiler must make sure to preserve it. The path p needs to be included because whether something is data-dependent can be path-dependent (in LLVM's

SSA based intermediate representation, this idea is made explicit through the use of phi nodes). This check is subtle and is discussed in Section 4.4.5.

The only thing that remains is to extend the cost function to take into account how we use dependencies. This proceeds by giving weights to $\overline{\mathsf{use\_data}}$ and $\overline{\mathsf{use\_ctrl}}$ and summing them up. Different weights should be given based on whether the dependencies are already present or need to be synthesized.

4.4.4 Scoped constraints

The one major thing lacking from the rules as presented so far is any consideration of `VEDGE_HERE` and `XEDGE_HERE`. Extending our system to handle scoped constraints is relatively straightforward. Recall that a scoped constraint between a and b establishes edges between executions of a and b, but only when the edges do not leave the scope of the constraint. Since we define the scope of a constraint in terms of the control flow graph, this means that a and b must be appropriately ordered along all control flow paths that do not pass through the binding site of the constraint.

With that in mind, the extensions to the rules are simple. For visibility (and push) edges, it is a simple matter of only requiring that we cut paths not containing the binding site:

$$\mathsf{vcut}(b,s,t) \;=\; \bigwedge_{s \xrightarrow{\mathsf{vo}} t@b} \mathsf{vcut}(b,s,t) \qquad \bigwedge_{p \in \mathsf{paths\_wo}(b,s,t)} \mathsf{vcut\_path}(p)$$

Here we write $s \rightarrow t@b$ to indicate a constraint edge from s to t that is bound at b and $\mathsf{paths\_wo}(b,s,t)$ to mean all simple paths from s to t without b in them. Non-scoped constraints will have a dummy b that does not appear in the CFG.

The modifications for execution edges are similar but have one additional wrinkle: when using control and data dependencies to ensure ordering, xcut_path can appeal to xcut and ctrl; we modify xcut_path to pass the binding site down to these. (In fact, this is the main use-case of scoped constraints: eliminating the need to order successive invocations when using data dependencies.) Since it can affect whether a data dependency exists along all path detours, we must also add a binding site argument to datacut_path that is passed down into can_data and

$\overline{\text{use\_data}}$.

$$
\begin{aligned}
& \bigwedge_{s \xrightarrow{\text{xo}} t @ b} \text{xcut}(b, s, t) \\
\text{xcut}(b, s, t) &= \bigwedge_{p \in \text{paths\_wo}(b,s,t)} \text{xcut\_path}(b, p) \\
\text{xcut\_path}(b, p) &= \text{vcut\_path}(p) \vee \\
& (\text{ctrlcut\_path}(p) \wedge \\
& \ (\text{ctrlcut}(b, s, s) \vee \text{xcut}(b, s, s))) \vee \\
& (\text{datacut\_path}(b, p) \wedge \text{xcut}(b, s, s)) \\
& (\text{where } s = \text{head}(p)) \\
\text{ctrlcut}(b, s, t) &= \bigwedge_{p \in \text{paths\_wo}(b,s,t)} \text{ctrl\_path}(p) \\
\text{datacut\_path}(b, p) &= \mathit{can\_data}(b, s, t, p) \wedge \overline{\text{use\_data}}(b, s, t, p) \\
& (\text{where } s = \text{head}(p), t = \text{tail}(p))
\end{aligned}
$$

4.4.5 Finding data dependencies

Why we need path dependence

Detecting whether two accesses can be ordered by a data-dependency in our setting is actually tricky. Consider the following motivating example:

```
int list_lookup(list_node_t **head) {
   XEDGE_HERE(load, load);
   XEDGE_HERE(load, use);

   list_node_t *node = L(load, *head);
   while (node != NULL) {
      if (L(use, node->key) == key) {
         return L(use, node->val);
      }
      node = L(load, node->next);
   }
   return -1;
}
```

Here we have a linked list being used as an association list mapping keys to values, and a function that searches through the list for a key. We want to avoid inserting any barriers at all in this code, and *just* use data dependencies. There are two key ordering constraints here. First, each load of a node pointer needs to execute after the loads of all the previous nodes in the list. Second, actual accesses to the data in a node needs to occur after the load that gave us the node pointer (though here we end up actually requiring that it occurs after all of the loads).

Properly compiling this example requires that our notion of data dependence be path dependent. To see why, consider this modified version of the above code in which the path-dependence

of values is made explicit using SSA phi nodes and the constraints are split up instead of taking advantage of name overloading:

```
int list_lookup(list_node_t **head) {
   XEDGE_HERE(head_load, next_load);
   XEDGE_HERE(next_load, next_load);
   XEDGE_HERE(head_load, use);
   XEDGE_HERE(next_load, use);

   node0 = L(head_load, *head);
   while (1) {
      node = Phi(node0, node1);
      if (node == NULL) break;

      if (L(use, node->key) == key) {
         return L(use, node->val);
      }
      node1 = L(next_load, node->next);
   }
   return -1;
}
```

Here we need to ensure that `next_load` $\xrightarrow{xo}$ `next_load`, and we want to use a data dependency. The snag, though, is that the loads of `node1` are *not* always data dependent on `node1`; sometimes it is dependent on `node0`, which comes from the load of the head. But in that scenario, we don't care: if we came in from outside the loop, then there isn't a previous execution of `next_load` that we need to be ordered with! So we need some way of taking into account the path of execution.

A naive (non-) solution and its problems

Cutting execution constraints is done on a per-simple-path basis, so one plausible-seeming and straightforward approach is to use this path information while looking for data-dependencies. That is, if we are trying to cut the `next_load` $\xrightarrow{xo}$ `next_load` edge, the only path we need consider is through the loop. On that path, `node` always takes the value from `node1`, and thus it is always data-dependent.

Unfortunately this strategy breaks down for more complicated examples, resulting in incorrect code generation. Consider a program (and an SSA-form version of it):

```
void test(rmc::atomic<int*> &a,
          int *p) {
   XEDGE_HERE(a, b);
   int *q = L(a, a);
   while (1) {

      int n = L(b, *q);
      q = p;
      foo(n);
   }
}
```

```
void test(rmc::atomic<int*> &a,
          int *p) {
   XEDGE_HERE(a, b);
   int *q0 = L(a, a);
   while (1) {
      int *q = Phi(q0, q1);
      int n = L(b, *q);
      int *q1 = p;
      foo(n);
   }
}
```

If we follow the above strategy and directly use simple path information to determine whether there is a data-dependency, we run into trouble here. There is only one simple path from `a` to `b`, and `n` depends on `q0` along that path, so the above strategy will report that it can be used to satisfy the execution edge. However, because of the later assignment `q = p`, loads in the later iterations of the loop are *not* data dependent on the original `q0` load; we actually need to insert a barrier!

Another similar broken example (and its SSA version) is:

```
void test(rmc::atomic<int*> &a,
          int *p, int i) {
   XEDGE_HERE(a, b);
   int *q = L(a, a);

   int *r;
   do {

      r = q;
      q = p;
   } while (--i);

   int n = L(b, *r);
   foo(n);
}
```

```
void test(rmc::atomic<int*> &a,
          int *p, int i0) {
   XEDGE_HERE(a, b);
   int *q = L(a, a);

   int *r;
   do {
      int i = Phi(i0, i1);
      r = Phi(q, p);
      int i1 = i - 1;
   } while (i1);

   int n = L(b, *r);
   foo(n);
}
```

In the only simple path from `a` to `b`, in which the loop executes exactly once, `n` depends on `q`. If the loop executes more than once (in which case the path has a cycle), it instead depends on `p` and so data dependency doesn't ensure ordering.

Note that this is a related but different difficulty than the one described earlier as "dependency trickiness" and solved by requiring execution ordering from an execution edge source to itself. There, given that we had a way to order a source action with all occurrences of its destinations that execute *before* the next time the source is executed, we sought to make sure that we can also properly order with destination actions that execute *after* the next source execution. Here we are concerned with making sure we satisfy the premise of that conditional, ensuring that we properly order our source with *all* executions of destination actions *before* the source action is executed again.

A solution

One approach to checking for data dependence is to abandon the path-centric nature and instead check that it holds on *all* paths, not merely each simple one. This can be done without enumerating all such paths (which would be impossible). To do this, we find all of the basic blocks reachable from the source (without passing through the binding site, if one is in use) and then trace backwards from the destination of potential data dependency through its operands, searching for the source while assuming that Phi nodes can take their values from any reachable basic block. The algorithm breaks down into three cases:

- If a regular instruction is of a type that can carry a data dependency (we support address calculations, casts, and memory loads—others could work as well), then it is data depen-

dent on some source instruction if at least one of its arguments is.

- A Phi-node instruction is data dependent on some source if, under the assumption that the Phi-node is data dependent on the source, *all* of its operands that come from basic blocks reachable from the source are data dependent on the source. The main subtlety here is that the check is done under the assumption that the Phi-node is data dependent: this is necessary because Phi-nodes will often depend on themselves in loops.
- Instructions are data-dependent on themselves.

Pseudocode is shown in Figure 4.1.

```
def dataDepSearch(dest, src, phis, reachable):
  if dest == src or dest in phis:
     return True
  elif canCarryDependency(dest):
     for op in operands(dest):
        if dataDepSearch(op, src, phis, reachable):
           return True
     return False
  elif isPhiNode(dest):
     for basicblock, op in phiOperands(dest):
        if (basicblock in reachable and
             not dataDepSearch(op, src, phis.union({dest}), reachable)):
           return False
     return True
  else:
     return False
```

Figure 4.1: Pseudocode for searching for a data dependency

While this strategy works, it is somewhat suboptimal. It abandons the path-centric behavior that everything uses and it makes the use of data-dependency all or nothing. For example, in the program:

```
void test(rmc::atomic<int*> &a,
          int *p, int i) {
   XEDGE_HERE(a, b);
   int *q = L(a, a);

   if (i) {
      process(i);
      q = p;
   }

   int n = L(b, *q);
   process(n);
}
```

```
void test(rmc::atomic<int*> &a,
          int *p, int i) {
   XEDGE_HERE(a, b);
   int *q0 = L(a, a);

   if (i) {
      process(i);
   }
   int *q = Phi(q0, p);

   int n = L(b, *q);
   process(n);
}
```

we would like to be able to use the data dependency from `q` to `n` in the case that the branch is not taken and execute a barrier when it is, but the approach described above doesn't allow us to detect that a data dependency exists only along one of the paths.

What we want is to in some way characterize complex paths using their underlying simple paths. Fortunately this turns out to be easy. The key insight is that every complex path can be viewed as a (not necessarily unique) simple path with "detour" cycles attached to it. Then, given a simple path, if the destination depends on the source along all complex paths that detour off the simple path, the data dependency will ensure execution ordering along that path.

To help compute whether this holds, we define the notion of the nodes reachable while traversing a simple path as being the nodes on the path along with any node reachable as part of a potential detour from the path. Armed with this notion, we can check data dependence along a path by checking, using the above algorithm, whether there is a dependency when Phi-nodes can take values from any basic block reachable along the path.

Finding the nodes reachable while traversing a path is quite simple: it is every node that belongs to the same strongly connected component as a node in the path. [8] Figure 4.2 shows the algorithm for using SCCs and path reachability to compute `can_data`.

```
def path_reachable(bindSite, path):
   sccs = SCCs of the control flow graph with bindSite removed

   reachable = {}
   for u in path:
      if u not in reachable:
         reachable = reachable.union(sccs[u])

   return reachable

def can_data(bindSite, src, dst, path):
   reachable = path_reachable(bindSite, path)
   return dataDepSearch(dest, src, {}, reachable)
```

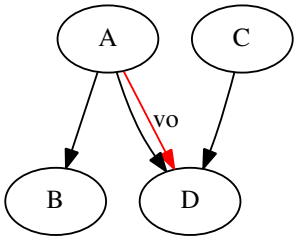
Figure 4.2: Pseudocode for using SCCs to restrict dependency search

4.4.6 Using the solution

While the process of using the SMT solution to insert barriers and take advantage of dependencies is fairly straightforward, there are a handful of interesting subtleties.

The first snag is that while our SMT problem thinks in terms of inserting barriers at control flow *edges*, we actually have to insert the barriers into the inside of basic blocks. This presents a snag when we are presented with control flow graphs like this:

[8] I felt very proud of myself for having developed a cute linear-time algorithm that computed it directly before Patrick Xia pointed this out to me.



The SMT solver will ask for a barrier to be inserted between basic blocks A and D, but there is a catch: the edge is a *critical edge* (a CFG edge where the source has multiple successors and the destination has multiple predecessors) for which there is no way to insert code along it without the code running on other paths as well. Fortunately, this is a well-known issue, so LLVM has a pass for breaking critical edges by inserting intermediate empty basic blocks. We require critical edges to be broken before our compilation pass runs, which means we can always safely insert the barrier at either the end of the source or the start of the destination.

This sort of barrier placement, which falls out naturally in our implementation of RMC, can't be achieved using C++11 memory orders on operations (though it could be achieved by manually placing C++'s low level thread fences).

The other snag comes when taking advantage of data and control dependencies: we need to make sure that later optimization phases can not remove the dependency. This ends up being a grungy engineering problem. Our current approach involves disguising the provenance of values involved in dependency chains to ensure that later passes never have enough information to perform breaking optimizations. This works, but is hacky, and it has the unfortunate side effect of sometimes preventing *correct* optimizations as well.

4.5 ARMv8

A recent revision of ARM, ARMv8, introduces new load and store instructions that have built in memory ordering behavior, likely to better support C++11 atomics [4], as well as a new barrier, `DMB LD`.

4.5.1 The ARMv8 memory model

The latest version of the ARM ARM [4] also provides, for the first time, a rigorously defined memory model. It also *strengthens* the memory model. While ARM traditionally allowed implementations to be truly non-multi-copy atomic, no hardware was ever implemented that did so. This allowed them to drop that allowance—ARM implementations must now be what they call "Other-multi-copy atomic": informally, if a write is observed by some processor *other* than the one who performed it, then it is observed by *all* other processors.

Interactions between processors is captured by the *Observed-by* relation, which relates two memory operations on *different* processors if they are related by reads-from, coherence order, or from-reads. [9]. Then the key relation *Ordered-before* is defined as the transitive closure of the

[9] A *from-read* edge between i and i' means that i reads from a write that is coherence-earlier than i'.

union of Observed-by and three other orders, the details of which we will skip: Dependency-ordered-before, Atomic-ordered-before, and Barrier-ordered-before.

Ordered-before is the key relation that restricts the behavior of programs, like *happens-before* in C++ and *visible-to* in RMC. Ordered-before is used to define the "external visibility requirement", which verifies that the *Observed-By* relation is consistent with *Ordered-Before* in an appropriate sense. While this is not how it is presented in the ARM manual, the "external visibility requirement" is equivalent to requiring that *Ordered-before* is acyclic [21].

Mapping RMC constructs into this new world order is relatively straightforward:

Execution order is mapped to Ordered-before, with the caveat that execution edges *from* writes have no meaning and are discarded.

Visibility order is also mapped to Ordered-before, with the caveat that edges from writes to reads are discarded, and one additional snag. If we have a visibility edge from a read r to a write w, and r reads from w', we requires w' to be visible before w. If w' is from another processor, then we will have w' Observed-by r, and thus w' Ordered-before w, and all is well. If w' was performed by the *same* thread, however, this does not hold! Thus, visibility order's mapping onto ARMv8 notions must additionally require that any local stores that could be read by r must also be Ordered-before w.

Push order is mapped directly to Ordered-before, with the same proviso about same thread writes as visibility.

4.5.2 Targeting ARMv8 release/acquire

ARMv8 adds the `LDA` and `STL` families of instructions, which they name "Load-Acquire" and "Store-Release". Despite these names, they are actually strong enough to implement C++11's SC atomics [41]: "Store-Releases" become visible before any subsequent "Load-Acquires" execute.

Store-Release writes appear after all program order previous reads and writes in Barrier-ordered-before. Since Barrier-ordered-before is a component of Ordered-before, Store-Releases may be viewed as being visibility-after all of their program order predecessors.

Load-Acquire reads, on the other hand, induce Barrier-ordered-before with all program order later reads and writes. This means that the Load-Acquire can be viewed as an execution edge to everything po-after it. It can *almost* also serve as a visibility edge, but not quite: it does not establish Ordered-before with a po-prior write if it reads-from it.

The changes needed to support Store-Release and and Load-Acquire instructions are fairly

noninvasive:

$$\begin{aligned}
\mathsf{vcut}(b,s,t) &= \left(\bigwedge_{p \in \mathsf{paths_wo}(b,s,t)} \mathsf{vcut_path}(p)\right) \vee \\
&\quad (is_write(t) \wedge \overline{\mathsf{use_rel}}(t)) \\
\mathsf{xcut}(b,s,t) &= \left(\bigwedge_{p \in \mathsf{paths_wo}(b,s,t)} \mathsf{xcut_path}(b,p)\right) \vee \\
&\quad (is_read(t) \wedge \overline{\mathsf{use_acq}}(t)) \vee \\
&\quad (is_simple_write(t) \wedge \overline{\mathsf{use_rel}}(t))
\end{aligned}$$

The functions is_read and is_write are inputs to the algorithm and check if an action is a read or a write, but do not require that the action be a "simple" read or write. That is, $\mathsf{is\_read}(s)$ is true if s is a read or a RMW, and likewise for is_write. The functions use_rel and use_acq are outputs and tell us whether their actions should be converted to release or acquire instructions.

As discussed above, an ARMv8 Load-Acquire establishes execution ordering with everything after it and a Store-Release establishes visibility ordering with everything before it. Thus, if the source of an edge can be made into an acquire or the destination into a release, this suffices to satisfy the edge. There is one subtlety to this, which is that RMW operations are implemented as a loop around Load-Exclusive and Store-Exclusive operations, not as a single fused operation. Making the Store-Exclusive part of the implementation of an RMW operation will ensure the correct visibility ordering with the write part, but does nothing to ensure that the read will see the correct values. Thus, Store-Release only satisfies an execution edge if the destination is a *simple* write. This is somewhat strange: it means that in some circumstances a visibility edge can be cut with a release while the execution edge that it implies can't.

4.5.3 Using dmb ld

ARMv8 introduces a new weaker variant of `dmb`, written `dmb ld`. The `dmb ld` barrier orders (using Barrier-ordered-before) all loads before the barrier before all stores and loads after it. Because all meaningful execution edges have a load as their source, this means that `dmb ld` can satisfy any execution edge with a straightforward barrier weaker than that needed for visibility, thus cleanly filling a niche that was left empty on POWER and ARMv7.

Supporting `dmb ld` in our SMT based system is extremely straightforward. An execution edge can be cut along a path by a `dmb ld` anywhere along it, so we simply add that to the long list of possible ways to cut a execution edge along a path:

$$\begin{aligned}
\mathsf{dmbld_path}(p) &= \bigvee_{e \in p} \overline{\mathsf{dmbld}}(e) \\
\mathsf{xcut_path}(b,p) &= \cdots \vee \mathsf{dmbld_path}(p)
\end{aligned}$$

4.5.4 Faking lwsync

Like Load-Acquire, `dmb ld` can *almost* serve as a visibility edge from earlier loads to later stores, but not quite: if the earlier load reads-from a po-prior read, the `dmb ld` does not establish Ordered-before between the po-prior write and subsequent writes.

When paired with a `dmb st`, however, `dmb ld` does become sufficient to ensure visibility ordering between earlier loads and later stores: if a prior load reads-from an earlier write on the same thread, then the `dmb st` will ensure that it is Ordered-before any later stores. Since `dmb st` already ensures visibility order from writes to writes, this means that a `dmb st; dmb ld` is sufficient to serve to satisfy any any visibility constraint. This makes `dmb st; dmb ld` very similar to POWER's `lwsync`. While `lwsync` has certain cumulativity properties I have not verified `dmb ld; dmb st` has, it does have all the properties the RMC compiler depends on.

Perhaps surprisingly, this scheme actually gives us better performance. Taking advantage of it is simple: when the SMT solution asks for an `lwsync` on ARMv8, we output `dmb ld; dmb st`.

Interestingly, the opportunity to soundly perform this optimization is new: before ARM was strengthened to be "Other multi-copy atomic", it would not have properly ensured visibility-order from loads to stores.

4.6 Compilation weights

A major lacuna in the above discussion is in the assignment of costs to all of the different operations. Our SMT based system is excellent at finding a solution that minimizes some quantity, but how should that quantity be defined?

A key difficulty here is that the entire optimization approach—assigning a fixed weight to each operation and scaling it based a guess of how often it will execute—is at best a coarse approximation and at worst fatally flawed. Modern processors are complicated beasts, and their memory systems doubly so.

x86

On x86 processors, we use the above SMT-based algorithm, but much of it is superfluous. We need to insert `MFENCE`s to resolve push edges and we need to prevent compiler reorderings that would violate execution and visibility constraints. We do this by disabling (notionally, by setting their cost to ∞) all mechanisms except sync and lwsync and inserting compiler barriers for lwsyncs.

The actual weights here are pretty irrelevant. It is important that `MFENCE`s be more expensive than compiler barriers, but that is about it. One `MFENCE` cannot replace multiple compiler barriers, nor vice versa, so the precise ratio of the weights makes no difference.

$$\begin{aligned} cost_{\mathsf{sync}} &= 800 \\ cost_{\mathsf{lwsync}} &= 500 \\ cost_{other} &= \infty \end{aligned}$$

ARMv7

$$
\begin{aligned}
cost_{\mathsf{sync}} &= 500 \\
cost_{\mathsf{lwsync}} &= \infty \\
cost_{\mathsf{dmbSt}} &= 350 \\
cost_{\mathsf{isync}} &= \infty \\
cost_{\mathsf{addCtrl}} &= 70 \\
cost_{\mathsf{useCtrl}} &= 1 \\
cost_{\mathsf{useData}} &= 1 \\
cost_{other} &= \infty
\end{aligned}
$$

ARM has no equivalent to Power's `lwsync`, having only the stronger `dmb`, which we represent with $cost_{\mathsf{sync}}$.

In our measurements, we found that using control dependencies plus `isync` was not an overall improvement, so we don't bother with it.

A single `dmb` can not substitute for multiple `dmb sts` or vice versa, so their relative cost is not by itself particularly important. The main question of relative costs, then, comes down to when to use a dmb in place of a collection of two or more newly inserted control dependencies and zero or more dmb sts. The question appears to be almost purely theoretical—not a single one of our case studies actually presents an opportunity to insert a new control dependency.

Using existing data and control dependencies needs to carry some cost, since the compiler tricks used to preserve them can inhibit optimizations, but our general model is that it should be very cheap. Actually *generating* new spurious data dependencies always struck me as super dubious, and when some performance testing using hand-created spurious dependencies yielded a substantial performance *degradation*, I happily took that as a reason to not implement it in the compiler.

ARMv8

$$
\begin{aligned}
cost_{\mathsf{sync}} &= 800 \\
cost_{\mathsf{lwsync}} &= 500 \\
cost_{\mathsf{isync}} &= \infty \\
cost_{\mathsf{dmbSt}} &= 350 \\
cost_{\mathsf{dmbLd}} &= 300 \\
cost_{\mathsf{addRel}} &= 240 \\
cost_{\mathsf{addAcq}} &= 240 \\
cost_{\mathsf{addCtrl}} &= 70 \\
cost_{\mathsf{useCtrl}} &= 1 \\
cost_{\mathsf{useData}} &= 1 \\
cost_{other} &= \infty
\end{aligned}
$$

Picking values on ARMv8 is much more meaningful than for the other platforms discussed. ARMv8's wide variety of operations provides that there are *many* opportunities for one sort of operation to substitute for multiple of another operation. For example, in some situations you could use either a `dmb ld` or multiple acquire loads. Or a `dmb` could be used in place of a `dmb ld` and a `dmb st`.

Note that `lwsync` here is really `dmb ld; dmb st`.

We tested a number of possible configurations, and settled on the above. Releases and acquires are generally preferred to inserting barriers, but not *vastly* so.

Power

On Power, we use the weights:

$$\begin{aligned} cost_{\mathsf{sync}} &= 800 \\ cost_{\mathsf{lwsync}} &= 500 \\ cost_{\mathsf{isync}} &= 200 \\ cost_{\mathsf{addCtrl}} &= 70 \\ cost_{\mathsf{useCtrl}} &= 1 \\ cost_{\mathsf{useData}} &= 1 \\ cost_{other} &= \infty \end{aligned}$$

Doing real testing on Power was something of an afterthought, so there is no reason to assign any real value to these numbers.

4.7 Sequentially consistent atomics

The core of our implementation of sequentially consistent atomics is almost abusively straightforward: `rmc::sc_atomic<T>` is simply a wrapper around `std::atomic<T>` that performs memory operations using `memory_order_seq_cst`. C++'s SC atomics provide the guarantees that we require—a total order, a visibility pre- edge for writes (release semantics), an execution post-edge for reads (acquire semantics)—so it is simple to take advantage of it.

Additionally, to avoid generating needlessly bad code in the cases where constraints are given to SC operations, actions consisting of just an SC operation are treated as already being release and/or acquire operations, which can be used to cut edges.

Several issues were recently discovered with common C++11 compilation schemes for sequentially consistent atomics, centered on mixing sequentially consistent accesses to a location with weaker accesses to the same location [30]. While the RMC core calculus allows this sort of mixing and gives semantics to it, RMC-C++ does not. This allows us to (for now, at least) deftly sidestep this issue.

Chapter 5

Case Studies

In order to evaluate the suitability of RMC as a practical method for implementing low-level concurrent programs, I implemented a collection of concurrent algorithms and data structures using both RMC and C++11 atomics.

A note (and some apologies?) about the case studies While RMC supports both C and C++, I choose to focus on C++ for all of the examples due mainly to the ability to have a proper `rmc::atomic<T>` type [1] and to load and store from atomic locations without writing clunky expressions like `rmc_load(&location)`. Once the decision to use C++ had been made, it seemed wrong to implement data structures that couldn't be used to store arbitrary data, and so I use templates to implement polymorphic versions of the data structures. After that I might have gone a bit overboard building some abstractions in a "C++y" style, which may have been a mistake. Additionally, since each data structure and memory management library has multiple versions (using different styles of weak memory), it was important to be able to automatically build all combinations of test+data structure+support library. Doing this involved some unfortunate hacks using the preprocessor and build system.

All of the case studies are available at `https://github.com/msullivan/rmc-compiler/case_studies`. We do not cover all of the implemented case studies here. Some are simply not very interesting (like readers-writer locks and condition variables) and some are skipped to avoid presenting multiple variants of the same algorithm.

5.1 Preliminaries

There are a few abstractions that will show up a number of times in the case studies.

[1] In RMC-C, we define the type `_Rmc(T)` as a macro the expands to a one element struct, but because struct definitions are nominal (even when not given a name!), `_Rmc(T)` types can't be used in function argument or return types without using typedef.

5.1.1 Generation-counted pointers

A common technique in lock-free data structure implementation is to associate a "generation count" with a pointer and to then use double-wide compare-and-swaps to modify the pointer and count at the same time. This is particularly useful in solving the "ABA problem", in which the reuse of an object could cause a CAS to succeed when correctness requires it to fail.

To support this idiom, our support library provides a `gen_ptr<T>` class. A `gen_ptr<T>` is an immutable object consisting of a `T` (which should be a pointer) and a generation count. Using tasteless C++ operating overloading, it can be dereferenced like an ordinary pointer. It has an `inc` method, such that `x.inc(p)` returns a new `gen_ptr<T>` that contains a pointer to `p` and a generation count that has been incremented by 1.

5.1.2 Tagged pointers

In many lock-free algorithms, it is necessary to store and extra bit or two of information alongside a pointer. This could be done using double-wide operations as above, but that is often wasteful. In practice, though, most objects will be at least 4-byte aligned, and so the low-order bits of a pointer can be used to store other data.

Our support library provides a `tagged_ptr<T>` class to do this. A `tagged_ptr<T>` is an immutable object that stores a tag bit in the low order bit of `T`, which should be a pointer. Using tasteless C++ operator overloading and implicit constructors, it can be used like an ordinary pointer and automatically coerced to as well. A second constructor provides a means of setting the tag and a `.tag()` method provides access.

5.2 Treiber stacks

Our first case study is one of the simplest concurrent data structures: the Treiber stack [45]. The core ideas of Treiber stacks are simple. To push, a new node is created that points to the current head; a compare-and-swap is performed to replace the head with the new node. If that fails, the process is repeated. To pop, the head node is read, its next pointer fetched, and the head compare-and-swapped with the next value. On failure, repeat.

Assuming that once allocated, stack nodes are only ever used as stack nodes, the one tricky part is handling the "ABA problem" that can arise when a node is removed from a stack and then reused. As described above, there is a problem. Imagine that thread 0 is popping from the stack and reads that the head is node A and the next node in the stack is node B. Then, other threads pop A and B and then push A back on top. Thread 0 then executes its CAS to replace A with B, which succeeds. But this is wrong, since B is no longer the next element on the queue. The simplest solution is to pair the pointer to top of the stack with a generation count that is updated by each operation. Then, the compare and swap to pop a node will only succeed when there have been no intervening operations on the stack.

5.2.1 RMC Version

We now present the core routines RMC version of generation counted Treiber stacks. These stacks require that nodes never be reused at another type but doesn't provide any facility for managing memory reuse. Indeed, Treiber stacks are the building block that we will use for building concurrent freelists. For brevity and clarity, we elide some uninteresting portions (constructors, accessors, forward method declarations).

Listing 5.1: `unsafe_tstack_gen_rmc.hpp`

```
template<typename T>
class UnsafeTStackGen {
public:
   struct TStackNode {
      rmc::atomic<TStackNode *> next_;
      T data_;

      /* ... */
   };
   /* ... */

private:
   rmc::atomic<gen_ptr<TStackNode *>> head_;
};

template<typename T>
void UnsafeTStackGen<T>::pushNode(TStackNode *node) {
   // Don't need edge from head_ load because 'push' will also
   // read from the write to head.

   VEDGE(node_setup, push);
   VEDGE(node_next, push);

   LPRE(node_setup);

   gen_ptr<TStackNode *> oldHead = head_;
   for (;;) {
      L(node_next, node->next_ = oldHead);
      if (L(push, head_.compare_exchange_weak(oldHead, oldHead.inc(node))))
         break;
   }
}

template<typename T>
typename UnsafeTStackGen<T>::TStackNode *UnsafeTStackGen<T>::popNode() {
   // We don't need any constraints going /into/ the pop; all the data
   // in nodes is published by writes into head_, and the CAS reads-from
   // and writes to head_, perserving visibility.
   XEDGE(read_head, read_next);
   XEDGE(read_head, out);

   gen_ptr<TStackNode *> head = L(read_head, this->head_);
   for (;;) {
```

```
        if (head == nullptr) break;
        TStackNode *next = L(read_next, head->next_);

        if (L(read_head, this->head_.compare_exchange_weak(
                head, head.inc(next)))) {
            break;
        }
    }

    LPOST(out);

    return head;
}
```

The primary invariant provided to users is that the data written into stack nodes will be visible to the code that reads them after a pop. More formally, we could require that Treiber stack push and pop operations be viewable as actions $push$ and pop, and that if $i \overset{\text{po}}{\rightarrow} push$, pop returns the node pushed by $push$, and $pop \overset{\text{po}}{\rightarrow} i'$, then $i \overset{\text{vt}}{\rightarrow} i'$. To more nicely handle interactions with other accesses, we adopt the stronger requirement that pushes and pops can be viewed as actions in which $\overset{\text{vis}}{\triangleleft} \# push$, $\overset{\text{exe}}{\triangleright} \# pop$, and if pop returns the node pushed by $push$, $push \overset{\text{vo}}{\rightarrow}{}^{+} pop$.

In this implementation, it is very simple to view pushes and pops as single actions: a push is the CAS that successfully adds the node and a pop is the CAS that successfully removes it. To match the spec, then, these need to have appropriate pre- and post- edges to the code before push and after post, respectively. Then, for the internals of the implementation, we also need to make sure that the values of the next pointer are correct. This requires pretty straightforward visibility and execution constraints on the accesses to the `next_` pointer. Because every write to `head_` is a RMW, every write to it is visibility ordered before every subsequent read from it, which satisfies the requirement about visibility order.

5.2.2 C++11 Version

Listing 5.2: `unsafe_tstack_gen_c11.hpp`

```
template<typename T>
void UnsafeTStackGen<T>::pushNode(TStackNode *node) {
    gen_ptr<TStackNode *> oldHead = head_.load(std::memory_order_relaxed);
    for (;;) {
        node->next_.store(oldHead, std::memory_order_relaxed);
        if (head_.compare_exchange_weak(
                oldHead, oldHead.inc(node),
                std::memory_order_release, std::memory_order_relaxed))
            break;
    }
}

template<typename T>
typename UnsafeTStackGen<T>::TStackNode *UnsafeTStackGen<T>::popNode() {
    gen_ptr<TStackNode *> head = this->head_.load(std::memory_order_acquire);
    for (;;) {
```

```
        if (head == nullptr) return nullptr;
        TStackNode *next = head->next_.load(std::memory_order_relaxed);

        // We would like to make the success order std::memory_order_relaxed,
        // but the success order can't be weaker than the failure one.
        // We don't need to release--because all of the writes to head_
        // are RMWs, every write to the head will be part of the
        // "release sequence" of all previous writes. Thus any read that
        // reads-from this will also synchronize-with any previous RMWs,
        // because this write is in their release sequence.
        if (this->head_.compare_exchange_weak(
              head, head.inc(next),
              std::memory_order_acquire, std::memory_order_acquire))
           break;
    }

    return head;
}
```

The invariant for the C++ version is simply that if pop returns the node pushed by $push$, then $push$ *synchronizes-with* pop.

This is achieved by having the CAS in push be a release and the CAS in pop be an acquire. Because all modifications to `head_` are done by RMW operations, all writes to `head_` after a push are in the *release sequence* of the push. Thus, since a pop is an acquire read from a write that is in the release sequence of the release write push, the push *synchronizes-with* the pop. Correctness of the actual `next_` pointers is ensured by using a release when push writes to head and an acquire when pop reads from it.

5.2.3 Discussion and Comparison

At first glance, the C++11 version seems to come out of this looking better, though I feel that this is somewhat superficial.

The first thing to note is that the RMC specification is much clunkier than the C++11 one. This is, however, easily fixed by an abbreviation. The pattern of being able to treat operations on a concurrent data structure as pre/post-tagged actions with implied visibility constraints will be an extremely common one.

We find ourselves, then, wanting to define a version of synchronizes-with for RMC. We say that i *synchronizes-with* i' if i has a visibility pre- edge, i' has an execution post- edge, and i is visibility-before i'. More formally:

$$i \overset{\text{sw}}{\to} i' \overset{\text{def}}{=} \overset{\text{vis}}{\lhd} \# i \wedge \overset{\text{exe}}{\rhd} \# i' \wedge i \overset{\text{vo}}{\to}{}^{+} i'$$

This doesn't actually *change* anything, but gives us an appropriate idiom to discuss the behavior of the program in a much more convenient way.

The C++11 version is clearly more concise than the RMC version. This is basically an unavoidable consequence of needing to use actual syntax to specify our constraints instead of actually being able to draw box and arrow diagrams. To the (limited and dubious) extent that such a comparison is meaningful, RMC with boxes and arrows feels to me more concise than

the C++11 version with explicit annotations. Of course, program concision is not the same as program legibility, or Perl would be more highly regarded.

One way to make the RMC version more concise would be to skip the fine-grained constraints and simply specify `VEDGE(pre, push)` and `XEDGE(read_head, post)`. This is a stronger constraint, but in practice should generate identical code. I believe the more explicit version is preferable, though. Given that the design and reasoning process for this code identified the two constraints separately, however, it is valuable for the code to reflect that. Indeed, this is nearly the entire premise of the RMC methodology: given that relative ordering constraints are a key reasoning principle, exposing them directly in the program aids program construction and understanding.

5.3 Epoch-based memory management

5.3.1 Introduction

Epoch-based reclamation [23] is an elegant technique in which threads must indicate when they are accessing locklessly protected data and can register objects to be reclaimed once all threads are guaranteed to no longer be using them. [2] Our implementation of epoch-based reclamation is heavily inspired by the Crossbeam library for Rust [46].

A client to the epoch library may "pin" the memory they might be accessing by entering an epoch critical section with code such as `auto guard = Epoch::pin();`. In a perhaps excessive bit of C++ism, `Epoch::pin()` returns a `Guard` object that manages the lifecycle of epoch critical sections. When a `Guard` goes out of scope, its destructor will leave the epoch critical section. Calling `guard.unlinked(p)` indicates that the object pointed to by `p` has been unlinked from any shared data structures and may be freed once no threads may be accessing them. Epoch critical sections may be nested.

5.3.2 Key epoch invariant

We write E_a to indicate an epoch entry, X_a to indicate the corresponding epoch exit, and M_a to indicate actions taken by user code during the epoch. We write $d(f_b)$ to indicate registering the actions f_b to run later in order to reclaim objects.

Then, given thread A executing E_a, M_a, X_a, and thread B executing E_b, M_b, X_b, where $d(f_b) \in M_b$, either $M_a \overset{\mathsf{vt}}{\to} f_b$ or $M_b \overset{\mathsf{vt}}{\to} M_a$. That is, either the actions in thread A's epoch critical section are visible to the object reclamations registered in thread B's critical section or the actions in thread B's critical section will be visible to the actions in thread A's.

5.3.3 Core stuff

While the core user-facing interface for Epoch-based memory management is built around `Epoch::pin()` and the `Guard` class and requires no user intervention to manage thread's entering

[2] Epoch reclamation is essentially a variant of RCU with an increased focus on efficient memory reuse.

and exiting the system, these are just convenience wrappers around the core internal interface. The core internal interface is built around the `Participant` class, which represents a single thread that is participating in the epoch-based memory management system.

The `Participant` class declaration is identical for both the RMC and C++11 versions. The type `epoch_atomic<T>` is defined to be either `std::atomic<T>` or `rmc::atomic<T>`, depending on which version is being compiled.

Listing 5.3: `epoch_shared.hpp`

```
// A Participant is a thread that is using the epoch
// library. Participant contains each thread's local state for the
// epoch library and is part of a linked list of
class Participant {
public:
   using Ptr = tagged_ptr<Participant *>;
private:
   // Local epoch
   epoch_atomic<uintptr_t> epoch_{0};
   // Nested critical section count.
   epoch_atomic<uintptr_t> in_critical_{0};

   // Next pointer in the list of epoch participants. The tag bit is
   // set if the current thread has exited and can be freed.
   epoch_atomic<Participant::Ptr> next_{nullptr};

   // Collection of garbage
   LocalGarbage garbage_;

   // List of all active participants
   static epoch_atomic<Participant::Ptr> participants_;

public:
   // Enter an epoch critical section
   void enter();
   // Exit an epoch critical section
   void exit();
   // Enter an epoch critical section, but don't try to GC
   bool quickEnter();
   // Attempt to do a garbage collection
   bool tryCollect();

   // Create a participant and add it to the list of active ones
   static Participant *enroll();
   // Shut down this participant and queue it up for removal
   void shutdown();

   void registerCleanup(GarbageCleanup f) {
      garbage_.registerCleanup(f);
   }
};
```

5.3.4 RMC Implementation

The core idea of epoch reclamation is that the system has a notion of the current "epoch", which is represented as an integer. When a thread enters an epoch critical section, it marks that it is in a critical section and records the current epoch. When a thread wants to register memory to be reclaimed, it registers the object in a list of data to free from the epoch that the thread is in. In order to free garbage, the global epoch must be *advanced*. This can only be done when every thread is observed to be either not in a critical section at all, or to be in the current epoch. When an epoch is advanced, the garbage from *two* epochs ago may be reclaimed.

The trickiest part of the code is the handling of handling of thread exiting, where we use the low-bit of a node's next pointer to indicate it has exited and needs to be cleaned up from the list.

As an optimization, we keep per-thread lists of garbage. A global list of garbage is maintained to handle garbage left over upon thread exit.

Listing 5.4: `epoch_rmc.cpp`

```
static rmc::atomic<uintptr_t> global_epoch_{0};

/////// Participant is where most of the interesting stuff happens
bool Participant::quickEnter() {
   PEDGE(enter, body);

   uintptr_t new_count = in_critical_ + 1;
   L(enter, in_critical_ = new_count);
   // Nothing to do if we were already in a critical section
   if (new_count > 1) return false;

   // Copy the global epoch to the local one;
   uintptr_t global_epoch = L(enter, global_epoch_);
   epoch_ = global_epoch;
   LPOST(body);
   return true;
}

void Participant::enter() {
   uintptr_t epoch = epoch_;
   if (!quickEnter()) return;

   // If the epoch has changed, collect our local garbage
   if (epoch != epoch_) {
      garbage_.collect();
   }

   if (garbage_.needsCollect()) {
      tryCollect();
   }
}

void Participant::exit() {
   // This is actually super unfortunate. quickEnter() will
   // subsequently write to in_critical_, and we need the body of a
```

```
   // critical section to be visible to code that reads from that
   // subsequent write as well as this one.
   // In C++11, we would make the write to in_critical_ be a release
   // and the convoluted release sequence rules would save us.
   VEDGE(pre, exit);
   LPOST(exit);
   uintptr_t new_count = in_critical_ - 1;
   L(__exit, in_critical_ = new_count);
}

bool Participant::tryCollect() {
   XEDGE(load_head, a);
   VEDGE(a, update_epoch);
   XEDGE(update_epoch, post);
   VEDGE(collect, update_local);

   uintptr_t cur_epoch = epoch_;

   // Check whether all active threads are in the current epoch so we
   // can advance it.
   // As we do it, we lazily clean up exited threads.
try_again:
   rmc::atomic<Participant::Ptr> *prevp = &participants_;
   Participant::Ptr cur = L(load_head, *prevp);
   while (cur) {
      Participant::Ptr next = L(a, cur->next_);
      if (next.tag()) {
         // This node has exited. Try to unlink it from the
         // list. This will fail if it's already been unlinked or
         // the previous node has exited; in those cases, we start
         // back over at the head of the list.
         next = Ptr(next, 0); // clear next's tag
         if (L(a, prevp->compare_exchange_strong(cur, next))) {
            Guard g(this);
            g.unlinked(cur.ptr());
         } else {
            goto try_again;
         }
      } else {
         // We can only advance the epoch if every thread in a critical
         // section is in the current epoch.
         if (L(a, cur->in_critical_) && L(a, cur->epoch_) != cur_epoch) {
            return false;
         }
         prevp = &cur->next_;
      }

      cur = next;
   }

   // Try to advance the global epoch
   uintptr_t new_epoch = cur_epoch + 1;
```

```
    if (!L(update_epoch,
          global_epoch_.compare_exchange_strong(cur_epoch, new_epoch))) {
       return false;
    }

    // Garbage collect
    LS(collect, {
       global_garbage_[(new_epoch+1) % kNumEpochs].collect();
       garbage_.collect();
    });
    // Now that the collection is done, we can safely update our
    // local epoch.
    L(update_local, epoch_ = new_epoch);

    return true;
}

// Participant lifetime management
Participant *Participant::enroll() {
    VEDGE(init_p, cas);

    Participant *p = L(init_p, new Participant());

    Participant::Ptr head = participants_;
    for (;;) {
       L(init_p, p->next_ = head);
       if (L(cas, participants_.compare_exchange_weak(head, p))) break;
    }

    return p;
}
void Participant::shutdown() {
    VEDGE(before, exit);
    LPRE(before);

    Participant::Ptr next = next_;
    Participant::Ptr exited_next;
    do {
       exited_next = Participant::Ptr(next, 1);
    } while (!L(exit, next_.compare_exchange_weak(next, exited_next)));

}
```

First, note that because of the visibility edges in `exit` and `tryCollect`, the bodies of epoch critical sections that have been observed to have been completed before advancing an epoch will be visible before the epoch update.

To see informally why the key epoch invariant holds, consider a thread A executing A executing E_a, M_a, X_a, and thread B executing E_b, M_b, X_b, where $d(f_b) \in M_b$. We want to show that either $M_a \overset{\text{vt}}{\to} f_b$ or $M_b \overset{\text{vt}}{\to} M_a$. Because of how deferred freeing actions are handled, f_b will be executed as part of some epoch entrance sequence E_c.

Epoch entrance sequences contain a push, which we will write as P_a for the push in E_a.

Since pushes are totally ordered, we will then have either $P_a \overset{\mathsf{vo}}{\to} P_c$ or $P_c \overset{\mathsf{vo}}{\to} P_a$. First, note that $M_b \overset{\mathsf{vo}}{\to} P_c$. Because the epoch must be advanced twice before f_b will be run, this means that M_b is visible before the epoch update write. Since E_c's read of the epoch is push-order before P_c, this means that if $P_c \overset{\mathsf{vo}}{\to} P_a$, we also have $M_b \overset{\mathsf{vt}}{\to} M_a$.

In the other case, in which $P_a \overset{\mathsf{vo}}{\to} P_c$, we first note that E_a's write to `in_critical_` will be visible to C. This means that in order to successfully GC, C needs to read either a subsequent write to `in_critical_` or to observe that A's epoch is the most up to date. If C reads a write from `in_critical_` when A leaves fully leaves a critical section (or a subsequent read), then we have that $M_a \overset{\mathsf{vo}}{\to} f_b$. In the other case, in which C observes that A is currently in the critical section started by E_a (or one it is nested in), C must observe A to have an up-to-date epoch in order to advance the epoch. Since, as discussed above, M_b is visible before the latest epoch update, and A's read of the epoch is push order before its body, this gives us $M_b \overset{\mathsf{vt}}{\to} M_a$. In the case where A's entrance performed an epoch update itself, A's read of the current epoch is merely execution ordered before the body, but this still establishes $\overset{\mathsf{vt}}{\to}$.

A stronger invariant

For some uses, such as the RCU protected linked lists we will discuss in Section 5.7, we will require epochs to satisfy a stronger invariant than the one presented above. In RCU-style uses, mutators may make modifications outside of an epoch critical section and then reclaim memory using epoch collection, which fits poorly into the stated invariant that only relates actions inside of epoch critical sections.

A stronger invariant can be described our earlier notion of synchronizes-with: Given thread A executing E_a, M_a, X_a, and thread B executing E_b, M_b, X_b, where $d(f_b) \in M_b$, either $M_a \overset{\mathsf{vt}}{\to} f_b$ or $X_b \overset{\mathsf{sw}}{\to} E_a$.

If—for the purposes of synchronizes-with—we consider an epoch entrance to be the push it contains and an exit to be the write, this follows pretty straightforwardly from the correctness argument above.

5.4 Michael-Scott queues

We now present a variant of Michael-Scott lock-free concurrent queues [38] that uses epoch collection system to manage memory. The adaptation of concurrent queues is based off the queue implementation in Crossbeam [46].

An MS queue maintains head and tail pointers for the queue. A dummy node is allocated so that these pointers are never null—they always point to a queue node. The "head" node, then, is actually a dummy node, and the first "real" node in the queue is the one after it.

The algorithm here is actually quite simple. For enqueue, we read out the tail and check its next pointer. Since the next pointer in the tail node and the tail pointer itself are updated separately, it is possible for the tail pointer to fall behind if a thread has enqueued a node but not yet updated the tail pointer. If the node *is* actually the tail, we attempt to enqueue it immediately. If not, we need to cope with the fact that the tail pointer is out-of-date. Since waiting for another thread to update it would compromise lock-freedom, we try to update the tail ourselves.

Dequeue is even more straightforward. As mentioned before, the first real node in the queue is the one *after* head, with the head itself being a dummy node. Thus we first read the head and the read its next pointer in order to find the real first node. If it is null, the list is empty and we fail. Otherwise we attempt to dequeue the node by compare-and-swapping the head to point to data node we are dequeuing, turning it into the dummy head.

Note that epoch collection is buying us quite a lot here. Much of the complexity in the original paper is in order to handle being able to reuse nodes in a freelist. There, pointers all need to carry generation counts and a number of additional consistency checks are added in order to properly handle nodes being moved between different queues. Additionally, without epoch collection, values must be read out of the node *before* doing the dequeuing compare-and-swap, which can cause trouble for large or complex data values.

5.4.1 RMC Version

Listing 5.5: `ms_queue_rmc.hpp`

```
template<typename T>
class MSQueue {
private:
   struct MSQueueNode {
      rmc::atomic<MSQueueNode *> next_{nullptr};
      optional<T> data_;
      // ...
   };

   rmc::atomic<MSQueueNode *> head_{nullptr};
   rmc::atomic<MSQueueNode *> tail_{nullptr};

   void enqueue_node(MSQueueNode *node);

public:
   MSQueue() {
      // Need to create a dummy node!
      head_ = tail_ = new MSQueueNode();
   }

   optional<T> dequeue();

   // ... enqueue() routines that take a T
};

template<typename T>
void MSQueue<T>::enqueue_node(MSQueueNode *node) {
   auto guard = Epoch::pin();

   // We publish the node in two ways:
   // * at enqueue, which links it in as the next_ pointer
   // of the list tail
   // * at enqueue_tail, which links it in as
   // the new tail_ of the queue
```

```
    // Node initialization needs be be visible before a node
    // publication is.
    VEDGE(node_init, enqueue);
    VEDGE(node_init, enqueue_tail);
    // Make sure to see node init, etc
    // This we should get to use a data-dep on!
    XEDGE(get_tail, get_next);
    // Make sure the contents of the next node stay visible
    // to anything that finds it through tail_, when we swing
    VEDGE(get_next, catchup_tail);


    // Marker for node initialization. Everything before the
    // enqueue_node call is "init".
    LPRE(node_init);

    MSQueueNode *tail, *next;

    for (;;) {
        tail = L(get_tail, this->tail_);
        next = L(get_next, tail->next_);

        // was tail /actually/ the last node?
        if (next == nullptr) {
            // if so, try to write it in.
            if (L(enqueue, tail->next_.compare_exchange_weak(next, node))) {
                // we did it! return
                break;
            }
        } else {
            // nope. try to swing the tail further down the list and try again
            L(catchup_tail, this->tail_.compare_exchange_strong(tail, next));
        }
    }

    // Try to swing the tail_ to point to what we inserted
    L(enqueue_tail, this->tail_.compare_exchange_strong(tail, node));
}

template<typename T>
optional<T> MSQueue<T>::dequeue() {
    auto guard = Epoch::pin();

    // Core message passing: reading the data out of the node comes
    // after getting the pointer to it.
    XEDGE(get_next, node_use);
    // Make sure we see at least head's init
    XEDGE(get_head, get_next);
    // Need to make sure anything visible through the next pointer
    // stays visible when it gets republished at the head or tail
    VEDGE(get_next, dequeue);
```

```
    MSQueueNode *head, *next;

    for (;;) {
        head = L(get_head, this->head_);
        next = L(get_next, head->next_);

        // Is the queue empty?
        if (next == nullptr) {
            return optional<T>{};
        } else {
            // OK, actually pop the head off now.
            if (L(dequeue, this->head_.compare_exchange_weak(head, next))) {
                break;
            }
        }
    }

    LPOST(node_use);

    // OK, everything set up.
    // next contains the value we are reading
    // head can be freed
    guard.unlinked(head);
    optional<T> ret(std::move(next->data_));
    next->data_ = optional<T>{}; // destroy the object

    return ret;
}
```

Like for Treiber stacks, the primary invariant provides to users is that the data written into queue nodes will be visible to the code that reads them after a dequeue. More formally, using the *synchronizes-with* abbreviation we defined while considering Treiber stacks, we could require that queue enqueue and dequeue operations be viewable as actions enq and deq, and that if deq returns the node added by enq, then $enq \overset{\text{sw}}{\rightarrow} deq$.

The core principle behind the visibility constraints here is that whenever a pointer to a node is written into the queue, the publishing write must be visibility-after all the initialization of the node. This is the reason for the fairly obvious `node_init` $\overset{\text{vo}}{\rightarrow}$ `enqueue` and `node_init` $\overset{\text{vo}}{\rightarrow}$ `enqueue_tail` constraints. It is also the reason for the `get_next` $\overset{\text{vo}}{\rightarrow}$ `catchup_tail` and `get_next` $\overset{\text{vo}}{\rightarrow}$ `dequeue`, which republish next pointers as tail and head pointers. Execution edges between reading a pointer to a node and reading data out of the node complete the message passing.

Also notable is the *absence* of any constraint from `get_next` to `enqueue`. It is important that, if `enqueue` succeeds, that it read from the most recent initialization of the node (and not from some earlier write). Because of our message passing, we know that `get_next` reads from either the most recent initialization or from an enqueuing write. If `next` is null, it must be the former. Then, because priority includes same-location program-order, the most recent initialization will be prior to `enqueue`.

5.4.2 C++11 Version

Listing 5.6: `ms_queue_c11.hpp`

```
template<typename T>
void MSQueue<T>::enqueue_node(MSQueueNode *node) {
   auto guard = Epoch::pin();

   MSQueueNode *tail, *next;

   for (;;) {
      // acquire because we need to see node init
      tail = this->tail_.load(std::memory_order_acquire);
      // acquire because anything we see through this needs to be
      // re-published if we try to do a catchup swing
      next = tail->next_.load(std::memory_order_acquire);

      // was tail /actually/ the last node?
      if (next == nullptr) {
         // if so, try to write it in.
         // release because publishing; not acquire since we don't
         // care what we see
         if (tail->next_.compare_exchange_weak(
               next, node,
               std::memory_order_release,
               std::memory_order_relaxed)) {
            // we did it! return
            break;
         }
      } else {
         // nope. try to swing the tail further down the list and try again
         // release because we need to keep the node data visible
         this->tail_.compare_exchange_strong(
            tail, next,
            std::memory_order_release,
            std::memory_order_relaxed);
      }
   }

   // Try to swing the tail_ to point to what we inserted
   // release because publishing
   this->tail_.compare_exchange_strong(
      tail, node,
      std::memory_order_release,
      std::memory_order_relaxed);
}

template<typename T>
optional<T> MSQueue<T>::dequeue() {
   auto guard = Epoch::pin();

   MSQueueNode *head, *next;
```

```
    for (;;) {
        head = this->head_.load(std::memory_order_acquire);
        next = head->next_.load(std::memory_order_acquire);

        // Is the queue empty?
        if (next == nullptr) {
            return optional<T>{};
        } else {
            // OK, actually pop the head off now.
            // release because we're republishing; don't care what we read
            if (this->head_.compare_exchange_weak(
                    head, next,
                    std::memory_order_release,
                    std::memory_order_relaxed)) {
                break;
            }
        }
    }

    // OK, everything set up.
    // next contains the value we are reading
    // head can be freed
    guard.unlinked(head);
    optional<T> ret(std::move(next->data_));
    next->data_ = optional<T>{}; // destroy the object

    return ret;
}
```

In this version, which uses only memory order annotations and not any fences (as is the generally recommended approach), all of the stores into the queue are releases and all of the loads are acquires. The compare-and-swap operations are treated just as stores, since we never care about the values read from them. This seems to be as weak as we can go using just annotations on memory operations.

We could weaken some operations by using `std::atomic_thread_fence`. In enqueue, the load from tail is only an acquire so that the node initialization will be published again if the tail pointer needs to be updated. Thus we could make the load from tail be relaxed and add an acquire fence before that compare-and-swap. Or the compare-and-swap itself could be relaxed and the fence made release-acquire. On the dequeue side, we could potentially do something similar.

Switching to using fences substantially complicates understanding and design of the code, however—generally the way that I figure out fence placement is by mentally running the RMC compilation algorithm! Additionally, whether or not the fences are better is likely to be platform specific.

The key sticking point here is that C++11 does not allow you to directly express the behavior required, merely different varieties of overly conservative approximations of it. RMC's ability to directly express subtle constraints like `get_next` $\overset{\text{vo}}{\rightarrow}$ `catchup_tail` is a strength that both makes the code clearer and allows better and more flexible code generation.

5.5 Queuing Spinlocks

In Section 2.1.3, we presented a simple implementation of spinlock[3] mutexes using compare-and-swap. These are somewhat unsatisfying, as they are not "fair" in any sense. A fair spinlock may be implemented quite simply using a fetch-and-add operation in a "ticket taking" scheme. This has an unfortunate scaling problem on many architectures, however: since every thread is monitoring the same location, there may be severe cache contention.

To avoid this cache contention, the Linux kernel has a variant of spinlocks called "qspinlocks" [16], based on prior work called "MCS locks" [37].

The core idea of qspinlocks is that a singly linked list of threads waiting for the lock is maintained. The lock itself is simply a pointer to the tail of the queue of waiting threads, with the low-bit of the pointer overloaded to indicate that the lock is currently unlocked. Each waiter structure in the queue consists of a pointer to the next element in the queue and a "ready" flag. After a thread adds itself to the tail of the waiting queue, it fills in the next pointer of the old tail, and then waits for the "ready" flag to indicate it is at the head of the queue, at which point it may start contending for the lock. Once a thread has acquired the lock, it sets the ready flag in its successor in the queue. Releasing the lock simply consists of clearing the low bit of the tail pointer.

Listing 5.7: `qspinlock_rmc.hpp`

```
class QSpinLock {
   struct Node {
      using Ptr = tagged_ptr<Node *>;
      rmc::atomic<Node *> next{nullptr};
      rmc::atomic<bool> ready{false};
   };

   rmc::atomic<Node::Ptr> tail_;


   void slowpathLock(Node::Ptr oldTail) {
      // makes sure that init of me.next is prior to tail_link in
      // other thread
      VEDGE(node_init, enqueue);
      // init of me needs to be done before publishing it to
      // previous thread also
      VEDGE(node_init, tail_link);
      // Can't write self into previous node until we have read out
      // the correct previous node (which we do while enqueueing).
      XEDGE(enqueue, tail_link);

      XEDGE(early_acquire, body);

      LS(node_init, Node me);
      Node::Ptr curTail;
```

[3]By "spinlock", we mean locks that simply spin in a loop until the lock can be acquired, rather than arranging for the thread to be descheduled. These are mostly appropriate inside the kernel itself, to be used when interrupts are disabled (though variants that repeatedly yield can be workable in userland).

```
bool newThreads;

// Step one, put ourselves at the back of the queue
for (;;) {
   Node::Ptr newTail = Node::Ptr(&me, oldTail.tag());

   // Enqueue ourselves...
   if (L(enqueue,
        tail_.compare_exchange_strong(oldTail, newTail))) break;

   // OK, maybe the whole thing is just unlocked now?
   if (oldTail == Node::Ptr(nullptr, 0)) {
      // If so, just try to take the lock and be done.
      if (L(early_acquire,
           tail_.compare_exchange_strong(
              oldTail, Node::Ptr(nullptr, 1))))
         goto out;
   }
}

// Need to make sure not to compete for the lock before the
// right time. This makes sure the ordering doesn't get messed
// up.
XEDGE(ready_wait, lock);
XEDGE(ready_wait, post); // XXX: PERFORMANCE HACK

// Step two: OK, there is an actual queue, so link up with the old
// tail and wait until we are at the head of the queue
if (oldTail.ptr()) {
   // * Writing into the oldTail is safe because threads can't
   // leave unless there is no thread after them or they have
   // marked the next ready
   L(tail_link, oldTail->next = &me);

   while (!L(ready_wait, me.ready)) delay();
}

// Step three: wait until the lock is freed
// We don't need a a constraint from this load; "lock" serves
// to handle this just fine: lock can't succeed until we've
// read an unlocked tail_.
while ((curTail = tail_).tag()) {
   delay();
}

// Our lock acquisition needs to be finished before we give the
// next thread a chance to try to acquire the lock or it could
// compete with us for it, causing trouble.
VEDGE(lock, signal_next);
XEDGE(lock, body);

// Step four: take the lock
```

```
      for (;;) {
         assert_eq(curTail.tag(), 0);
         assert_ne(curTail.ptr(), nullptr);

         newThreads = curTail.ptr() != &me;

         // If there aren't any waiters after us, the queue is
         // empty. Otherwise, keep the old tail.
         Node *newTailP = newThreads ? curTail : nullptr;
         Node::Ptr newTail = Node::Ptr(newTailP, 1);

         // This can fail if new threads add themselves to the
         // queue. However, nobody else can actually *take* the
         // lock, so we'll succeed quickly.
         if (L(lock, tail_.compare_exchange_strong(curTail, newTail))) break;
      }

      // Step five: now that we have the lock, if any threads came
      // in after us, indicate to the next one that it is at the
      // head of the queue
      if (newThreads) {
         // Next thread might not have written itself in, yet,
         // so we have to wait.
         // Waiting for threads *after* you in the queue kind of
         // offends me, to be honest.
         Node *next;
         XEDGE(load_next, signal_next);
         XEDGE(load_next, post); // XXX: PERFORMANCE HACK
         while (!L(load_next, next = me.next)) delay();
         L(signal_next, next->ready = true);
      }

   out:
      LPOST(body);
      return;
   }

public:
   void lock() {
      XEDGE(lock, post);

      // If the lock is unlocked and has no waiters, we can acquire
      // it with no fanfare. Otherwise we need to fall back to the
      // slow path.
      Node::Ptr unlocked(nullptr, 0);
      if (!L(lock,
           tail_.compare_exchange_strong(unlocked, Node::Ptr(nullptr, 1)))){
         slowpathLock(unlocked);
      }
      LPOST(body);
   }
   void unlock() {
```

```
        VEDGE(pre, unlock);
        LS(unlock, clearLowBit(tail_));
    }

};
```

In this code, we experiment with a different style than most of the other RMC code: instead of placing all of the annotations at the top of the function, we place them throughout the function, nearer to the binding sites of the labels. I'm not quite sure which I prefer.

The core lock and unlock routines have the typical pre and post edges that are standard for mutexes, while most of the interesting work is in `slowpathLock`. While there are a lot of constraints, most of them are fairly straightforward message passing. There is one interesting subtlety in some of the message passing, however: `node_init` $\overset{\text{vo}}{\rightarrow}$ `enqueue` and `enqueue` $\overset{\text{xo}}{\rightarrow}$ `tail_link` form a message passing pair that ensures that the old tail's node initialization is visible to the new tail updating the old tail's next pointer. The subtlety here is that the new tail does not care one bit about the values in the node; it never reads them. What does matter, however, is that the new tail's update is coherence-later than the node initialization! Similar circumstances obtain for the the `node_init` $\overset{\text{vo}}{\rightarrow}$ `tail_link` and `load_next` $\overset{\text{xo}}{\rightarrow}$ `signal_next` pair.

The `delay()` function is intended as a placeholder for architecture-specific instructions intended to help optimize busy-waiting, such as the x86 `PAUSE` instruction. In the actual measured code, the function is empty and does not actually delay anything (though see below).

The post edges marked `PERFORMANCE HACK` are unfortunate. In our performance testing, tight busy waits seem to be a major counterexample to `rmc-compiler`'s assumption that it is best to execute as few barriers as are necessary semantically. In this sort of tight, unproductive, busy wait (where no attempt to progress is being made, we are merely waiting on the contents of a memory location), we actually obtain *substantially* better ARMv7 performance by executing a barrier immediately after every read. [4] To enforce this, we add a post edge to our unproductive busy waits. This is, of course, a big bummer: one of RMC's promises is that it ought to handle this sort of thing for you. Likely some sort of heuristic for busy waits could be added to `rmc-compiler`, though it does not seem urgent: actual unproductive busy waiting is typically restricted to a small number of spinlock implementations in the kernel. Userspace locks ought not busy wait (my examples to the contrary being for demonstration purposes only) and the definition of lock-freedom precludes busy waiting in lock-free algorithms.

5.6 Sequence locks

Sequence locks (seqlocks) are yet another cute approach for protecting read-mostly data [24]. The central idea is that the presence of readers in a critical section never blocks writers. Instead, readers are informed upon unlock time if a writer used the lock during the critical section, allow-

[4] Some of the effect appears to come from simply slowing down the busy-waiting thread and thus (presumably) reducing cacheline contention—making `delay()` actually perform a delay loop does increase performance of an implementation that places the barrier after the loop. Nevertheless, executing a barrier after each read in the loops still yielded the best performance. Sigh.

ing the reader to retry. While this is too weak for many applications, it is well suited for read-side critical sections that only need to copy several pieces of data in a coherent manner. (A variant of this is used in Linux to protect multiword timestamps exposed by the kernel to userspace.)

Reader-side client code, then, looks something like:

```
for (;;) {
   auto tag = lock.read_lock();
   x = foo.x;
   y = foo.y;
   if (lock.read_unlock(tag)) break;
}
```

Seqlocks have fiddly interactions with language memory models. Hans Boehm analyzes them in the context of C++11 in a paper entitled "Can Seqlocks Get Along With Programming Language Memory Models?" [11]. The first snag is that because read-side seqlock critical sections are not protected against writes, there are data races on the locations protected by the seqlock. This means that, in C++11 and RMC, at least, these locations must be atomic locations in order to avoid introducing undefined behavior. This is somewhat unfortunate, but there is nothing to do about it. For efficiency and interface ease of use, we want to impose no restrictions on how the client code uses those locations. That is, we shouldn't require any particular memory ordering in C++11 nor should we require that the client specify any ordering constraints for them in RMC.

5.6.1 RMC version

The core idea of the sequence lock implementation is quite simple. A sequence lock is simply a single integer. The low-order bit of the integer indicates whether the lock is held by a writer. Writers increment the lock on both lock and unlock. A read lock consists of simply snapshotting the value of the lock, while a write lock must check if a retry is necessary. If the lock was held at the start or the the value of the lock has changed, then there has been interference and the critical section must be retried.

Listing 5.8: `seqlock_rmc.hpp`

```
class SeqLock {
private:
   rmc::atomic<uintptr_t> count_{0};
   bool is_locked(uintptr_t tag) { return (tag & 1) != 0; }

public:
   using Tag = uintptr_t;

   Tag read_lock() {
      // Lock acquisition needs to execute before the critical section
      XEDGE(read, post);
      return L(read, count_);
   }

   bool read_unlock(Tag tag) {
      // The body of the critical section needs to execute before
```

```
        // the unlock check, because if we observed any writes from a
        // writer critical section, it is important that we also
        // observe the lock.
        XEDGE(pre, check);
        return !is_locked(tag) && L(check, count_) == tag;
    }

    void write_lock() {
        // Lock acquisition needs to execute before the critical
        // section BUT: This one is visibility! If a reader observes a
        // write in the critical section, it is important that it also
        // observes the lock.
        VEDGE(acquire, out);

        for (;;) {
            Tag tag = count_;
            if (!is_locked(tag) &&
                L(acquire, count_.compare_exchange_weak(tag, tag+1))) {
                break;
            }
            delay();
        }
        LPOST(out);
    }

    void write_unlock() {
        VEDGE(pre, release);
        uintptr_t newval = count_ + 1;
        L(release, count_ = newval);
    }
};
```

The constraint annotations in `read_lock` and `write_unlock` are totally unremarkable: an execution post-edge from the read in `read_lock` and a visibility pre-edge in the write in `write_unlock`. This is exactly what we normally expect in a lock implementation.

The other functions are a bit more unusual. To see why, note that a key property of is that if a read-side critical section observes writes from a write-side critical section that didn't complete before the read-side section began, the read-side critical section must fail. Thus, it is important that if the read-side critical section body reads a write from a write-side critical section that the lock acquisition be visible to the `read_unlock` at the end of the critical section. Doing this has two parts: first, `write_lock`'s acquisition of the lock is made visibility-before all code in the write-side critical section. This means that it is visible before any modifications made. Second, the body of the read-side critical section is execution-before the check of the lock value. Combined, this means that if a critical section is interfered with, the read unlock will fail.

5.6.2 C++11 Version

Listing 5.9: `seqlock_c11.hpp`

```
class SeqLock {
```

```
private:
   std::atomic<uintptr_t> count_{0};

   bool is_locked(uintptr_t tag) { return (tag & 1) != 0; }

public:
   using Tag = uintptr_t;

   Tag read_lock() {
      return count_.load(std::memory_order_acquire);
   }

   bool read_unlock(Tag tag) {
      // Acquire fence is to ensure that if we read any writes
      // from a writer's critical section, we also see the writer's
      // acquisition of the lock.
      std::atomic_thread_fence(std::memory_order_acquire);
      return !is_locked(tag) && count_.load(std::memory_order_relaxed) == tag;
   }

   void write_lock() {
      for (;;) {
         Tag tag = count_.load(std::memory_order_relaxed);
         if (!is_locked(tag) &&
            count_.compare_exchange_weak(tag, tag+1,
                                 std::memory_order_relaxed)) {
            break;
         }
         delay();
      }
      // Release fence ensures that anything that sees our critical
      // section writes can see our lock acquisition.
      // Acquire fence so that the CAS above acts as an acquire
      // in the usual way.
      std::atomic_thread_fence(std::memory_order_acq_rel);
   }

   void write_unlock() {
      uintptr_t newval = count_.load(std::memory_order_relaxed) + 1;
      count_.store(newval, std::memory_order_release);
   }
};
```

Here, `read_lock` and `write_unlock` use acquire and release operations, respectively, as is normal. The critical section interference problem is solved by resort to C++11 much-maligned explicit fences (nominally intended for backward compatibility). In order to ensure that if a read-side critical section observes writes from a write-side critical section, `read_unlock` will observe the `write_lock`, we use an acquire fence in read lock release and a release fence in write lock acquisition. Normally we would make the compare-and-swap in `write_lock` an acquire operation, but as a (probable) optimization we instead make the fence an acquire fence as well.

5.6.3 Comparison

RMC comes out looking quite a bit better here, I think. While presenting a similar C++11 implementation, Boehm is not happy with it. He remarks, of fences, "we tend to discourage their use, since they tend to be very subtle to use correctly" and that the fence based solution "appears exceptionally unnatural to us." While the RMC implementation has some noteworthy parts, it is overall a relatively straightforward application of the usual tools.

Boehm also criticizes the fence-based version for overconstraining the memory ordering—it "prevents later operations in the reader thread from moving *into* the reader critical section". The RMC version does not have this weakness; actions in the body of a read-side critical section are execution ordered before *only* the load of the lock's count, not before all subsequent code.

Boehm suggests another potential approach, which is to make the load in `read_unlock` be a "read-dont-modify-write"
operation of the form `count_.fetch_add(0, std::memory_order_release)` and relying on the total order of RMWs to a location for correctness. This works, but requires compiler support for optimizing "read-dont-modify-write" operations into reads and fences, which strikes me as fragile and silly.

5.7 RCU-protected linked lists

Read-copy-update, or RCU, is an approach to protecting read-mostly data structures that optimizes for extremely efficient read-side accesses [34]. The standard interface for RCU includes routines for entering and exiting "read-side critical sections", a routine (`synchronize_rcu`) that allows a writer to wait until all "currently" executing read-side critical sections have completed, and a routine (`call_rcu`) to register a callback that will be run after "currently" executing read-side critical sections. We remarked early that epoch-based reclamation is essentially a variant of RCU with a focus on efficient memory reuse—that is, on efficient implementation of `call_rcu`. It is easy to add `synchronize_rcu` to our epoch library, though not necessary for this example.

A common application of RCU is protecting read-mostly linked lists. In this scheme, writers use a mutex to protect modifications to the list, but readers merely enter a read-side critical section. When writers remove elements from the list, they wait until any readers that may have a pointer to the node have finished their critical section before freeing the object. Since writers can modify the data concurrently with readers accessing it, care must be taken to to ensure that if a reader sees a node in the list, it sees properly initialized data in the object. With some care, this can be done with no read-side memory barriers apart from what is necessary to implement RCU itself.

5.7.1 RMC Version

For our example, we implement a list of `widget` objects. A `widget` contains a key and two value objects, as well as the linked list fields. The `next` pointer is atomic because readers will use it to perform list traversals, potentially concurrently with modifications. The `prev` pointer is used

only by writers, to simplify list manipulations, and so need not be atomic. We use a circularly linked list in which a dummy `widget` object is used as the list head. [5]

Listing 5.10: `rculist_user_rmc_simple.hpp`

```
struct widget {
   unsigned key{0};
   unsigned val1{0}, val2{0};

   rmc::atomic<widget *> next{nullptr};
   widget *prev{nullptr};

   widget(unsigned pkey, unsigned pval1, unsigned pval2) :
      key(pkey), val1(pval1), val2(pval2) {}
   widget(widget *n, widget *p) : next(n), prev(p) {}
};

struct widgetlist {
   // A dummy widget is used as the head of the circular linked list.
   widget head{&head, &head};
   std::mutex write_lock;
};

widget *widget_find_fine(widgetlist *list, unsigned key);
widget *widget_find(widgetlist *list, unsigned key);
void widget_insert(widgetlist *list, widget *obj);
```

And now, the implementation code:

Listing 5.11: `rculist_user_rmc_simple.cpp`

```
// Perform a lookup in an RCU-protected widgetlist, with
// execution edges drawn to the LGIVE return action.
// Must be done in an Epoch read-side critical section.
widget *widget_find_fine(widgetlist *list, unsigned key) {
   XEDGE_HERE(load, load);
   XEDGE_HERE(load, use);
   widget *node;
   widget *head = &list->head;
   for (node = L(load, head->next); node != head; node = L(load, node->next)) {
      if (L(use, node->key) == key) {
         return LGIVE(use, node);
      }
   }
   return nullptr;
}
```

[5] We also have a generic implementation of intrusive RCU-protected linked lists in the Linux style that works by embedding `rculist_node` fields into objects and having most central list operations operate on `rculist_node` objects. A rather frightening macro allows the recovery of pointers to containing objects from pointers to `rculist_node`s. To avoid this extra complexity, we present this more specialized code instead. (The rather frightening macro is a C-style approach to this. In C++, we can instead use rather frightening templated code, possibly combined with inheritance.)

```
// Perform a lookup in an RCU-protected widgetlist with a traditional
// post edge.
// Must be done in an Epoch read-side critical section.
widget *widget_find(widgetlist *list, unsigned key) {
   XEDGE(find, post);
   return L(find, widget_find_fine(list, key));
}

static void insert_between(widget *n, widget *n1, widget *n2) {
   VEDGE(pre, link);
   n->prev = n1;
   n2->prev = n;
   n->next = n2;
   L(link, n1->next = n);
}
static void insert_before(widget *n, widget *n_old) {
   insert_between(n, n_old->prev, n_old);
}
static void replace(widget *n_old, widget *n_new) {
   insert_between(n_new, n_old->prev, n_old->next);
}

// Insert an object into a widgetlist, replacing an old object with
// the same key, if necessary.
// Must *not* be called from an Epoch read-side critical section.
void widget_insert(widgetlist *list, widget *obj) {
   list->write_lock.lock();
   // We needn't give any constraints on the node lookup here. Since
   // insertions always happen under the lock, any list modifications
   // are already visible to us.
   widget *old = widget_find_fine(list, obj->key);

   // If nothing to replace we just insert it normally
   if (!old) {
      insert_before(obj, &list->head);
      list->write_lock.unlock();
      return;
   }

   replace(old, obj);
   list->write_lock.unlock();

   // Register node to be reclaimed
   auto guard = Epoch::pin();
   guard.unlinked(old);
}
```

The core internal routine for inserting a node into a list, `insert_between`, contains the one constraint needed here on the writer side: a visibility pre-edge before updating a next pointer. This ensures the visibility of object initialization to readers traversing the list. There are no writer-writer visibility concerns, since write access is protected by a mutex.

The main routine for reader use is `widget_find_give`, which searches a `widgetlist` for a

node with a particularly `key`. In order to support fine-grained constraints, all of its loads from list pointers are execution-before its `LGIVE` return action. Client code can take advantage of this to draw fine-grained edges from the `widget_find_give` call to accesses of fields in the list. There are two constraints in `widget_find_give` itself: first, each load of a `next` pointer is execution before all subsequent `next` pointer loads. Second, `next` pointers loads are execution before all uses of `widget` data elements—in particular, the load of the `key` and the `LGIVE` return action. Together, these ensure that readers see a consistent list and avoid non-atomic data races. All of the constraints can be implemented simply by taking advantage of the data dependencies in the code.

Some client code, then, might look like:

Listing 5.12: `rculist_user-test.cpp`

```
void consume(Test *t, unsigned key) {
   XEDGE_HERE(find, a);
   auto guard = Epoch::rcuPin();

   widget *nobe = LTAKE(find, widget_find_fine(&t->widgets, key));
   assert(nobe);
   assert_eq(key, L(a, nobe->val1) - L(a, nobe->val2));
}
```

Here, `Epoch::rcuPin()` is a routine that enters an epoch critical section without doing garbage-collection related checks (by calling `Participant::quickEnter()`). This is useful for workloads like this in which readers perform no memory reclamation.

One clear downside of this technique, of course, is that it leaks into client code, where it is quite syntactically heavyweight. The `widget_find` routine offers a potential middle ground: fine-grained edges are used internally during the list search, but a coarse-grained edge establishes the interface with client code. In testing on on ARMv7 machine, the lion's share of the (substantial!) performance improvements obtained by using data dependencies on this test came from eliminating `dmb`s inside the list traversal, with the elimination of the final `dmb` only a slight improvement.

Macro based traversal

The code for list traversal above is somewhat painful. If list traversal will appear in multiple places, it would be profitable to use a list traversal macro such as:

Listing 5.13: `rculist_user_rmc_simple_macro.cpp`

```
#define rculist_for_each2(node, h, tag_load, tag_use) \
   XEDGE_HERE(tag_load, tag_load); XEDGE_HERE(tag_load, tag_use); \
   for (node = L(tag_load, (h)->next); \
        node != (h); \
        node = L(tag_load, node->next))
#define rculist_for_each(node, head, tag_use) \
   rculist_for_each2(node, head, __rcu_load, tag_use)

widget *widget_find_fine2(widgetlist *list, unsigned key) {
   widget *node;
```

```
    rculist_for_each(node, &list->head, r) {
        if (L(r, node->key) == key) {
            return LGIVE(r, node);
        }
    }

    return nullptr;
}
```

Here, we introduce a macro `rculist_for_each(node, head, tag_use)` that iterates over the list `head` using `node` as its element pointer. Additionally, all `next` pointers loads during the traversal will be execution-before the tag `tag_use`.

5.7.2 C++11 Version

To produce a C++11 version of RCU-protected linked lists, we would make the `link` store `memory_order_release` and the `load` read `memory_order_consume`. Other atomic accesses can be `memory_order_relaxed`. As discussed in Section 1.3.2, consume establishes a happens-before relationship from the releasing write to actions that are data-dependent on the consume load. Since taking advantage of the fine-grained data-dependent based ordering does not require any explicit work on the client side, it probably suffices to *only* provide a fine-grained consume-based interface. That said, a more coarse grained interface may be built on top of the fine-grained one using `std::atomic_thread_fence(std::memory_order_acquire)`.

Unfortunately for the C++11 implementation, `memory_order_consume` is not properly implemented in any known C++ compiler. To avoid the painful and often intractable task of tracking down all possible dependencies, compilers simply treat consume the same as acquire, rendering all of this moot.

The Linux kernel, where use of RCU-protected data structures is widespread, takes advantage of the ordering provided by data dependencies without using the unimplemented consume feature. They confront the lack of assurances from the language by instead attempting to quantify over all possible compiler transformations, carefully documenting a long list of things to avoid doing, lest the compiler optimize away a crucial data dependency [33]. While this approach is fraught with danger, it has served extremely well from a performance perspective.

5.8 An undergrad course project

One difficulty in evaluating all these examples is that they were written by one of the designers of RMC. Doing a proper user study would be infeasible, as the amount of background and study required to reasonably work with weak-memory systems is substantial. We were, however, able to collect some decent anecdotal data from two students who elected to do a projecting using RMC.

As a final project for an undergraduate course on parallel programming, Sam Bowen and Josh Zimmerman implemented two lock-free data structures using both RMC and C++11 [15]. They implemented lock-free stacks and a lock-free linked lists, reporting, on ARM, a 2.01x speedup relative to C++11 for stacks and 1.21x for linked lists. This is encouraging, though they also

report 1.34x and 1.22x speedups on x86, which hints that the C++11 code is probably pretty bad. Looking at the code confirms this, unfortunately—it is possible that the authors believed that `atomic_load` in C behaves as if the memory order was `memory_order_relaxed`, rather than `memory_order_seq_cst`. This is arguably still a point in our favor, however, if RMC is easier to learn well enough to write efficient code.

Bowen and Zimmerman preferred RMC to C++11 atomics, reporting that "using [RMC] was significantly more straightforward than manually using the C11 atomic intrinsics." The main complaints they had were the lack of a tool to visualize potential behaviors and the handful of compiler bugs they encountered (which were all promptly fixed).

One major bias in all this, though, is that RMC was explicitly the focus of the assignment, and so more time and energy was probably put into the RMC versions. Additionally, one of the students was a friend of mine, and may have wanted me to feel good.

Overall, though, the result of this project was encouraging: the students were able to learn how to use and reason about RMC and preferred it to the alternatives.

Chapter 6

Performance Evaluation

6.1 Generated code performance

To evaluate `rmc-compiler`, we implemented a number of low-level concurrent data structures using C++ SC atomics, C++ low-level atomics, and RMC and measured their performance on ARMv7, ARMv8, and Power. We performed our ARMv7 tests on an NVIDIA Jetson TK1 quad-core ARM Cortex-A15 board and our ARMv8 tests on an ODROID-C2 quad-core ARM Cortex-A53 board. Power tests were performed on an IBM Power System E880 (9119-MHE). Tests were compiled using Clang/LLVM 3.9.

Michael-Scott queues

In Figure 6.1 we show the results of testing a number of implementations of Michael-Scott concurrent queues [38]. The "freelist" tests are traditional Michael-Scott queues that use generation counts and double-word compare-and-swaps to avoid the ABA problem and a Treiber stack [45] as a free list for queue nodes (which may be reused as nodes in other concurrent queues but not as other types of objects). The Treiber stacks used in the freelist are those discussed in Section 5.2. The "epoch" tests use the epoch-based reclamation [23] system discussed in Section 5.3. The RMC epoch version is that presented in Section 5.4. For these tests, though, we use freelist and epoch libraries implemented using C++11 atomics.

Both queue variants are tested using a variety of workloads: single-producer/single-consumer, single-producer/multiple-consumer, multiple-producer/multiple-consumer, and four threads alternating between enqueues and dequeues. The "aggregate" is the geometric mean of the other test results, meant to provide a rough aggregate measurement.

On ARMv7, across all of the queue workloads, the RMC version performs best by a modest but significant amount. As expected, the C++11 low-level atomic version outperforms the SC atomics version. The performance wins for RMC are greater for the freelist variant than for the epoch variant, likely because the additional complexity of the freelist version gives RMC more room to work.

The differences on ARMv8 are much more modest. RMC very slightly wins, but it is a close thing. Tellingly, SC atomics perform nearly as well. As discussed before, ARMv8's new "Release" and "Acquire" instructions are misnomers, and actually can be used to implement C++'s

sequentially consistent atomics. On ARMv8, then, SC atomics are as efficient as release/acquire ones.

On Power, RMC comes out on top, by a small margin.

Treiber Stacks

In Figure 6.2 we show the results of testing a number of implementations of Treiber concurrent stacks (the C++11 and RMC versions of which were shown in Section 5.2. The categories of test are the same as for queues.

Here, on ARMv7 the results are somewhat more mixed and workload dependent: the epoch based RMC stack wins on most tested workloads, but loses slightly on a single-producer/multiple-consumer workload. RMC fares somewhat less well for freelist based stacks, suffering two slight loses and two slight wins.

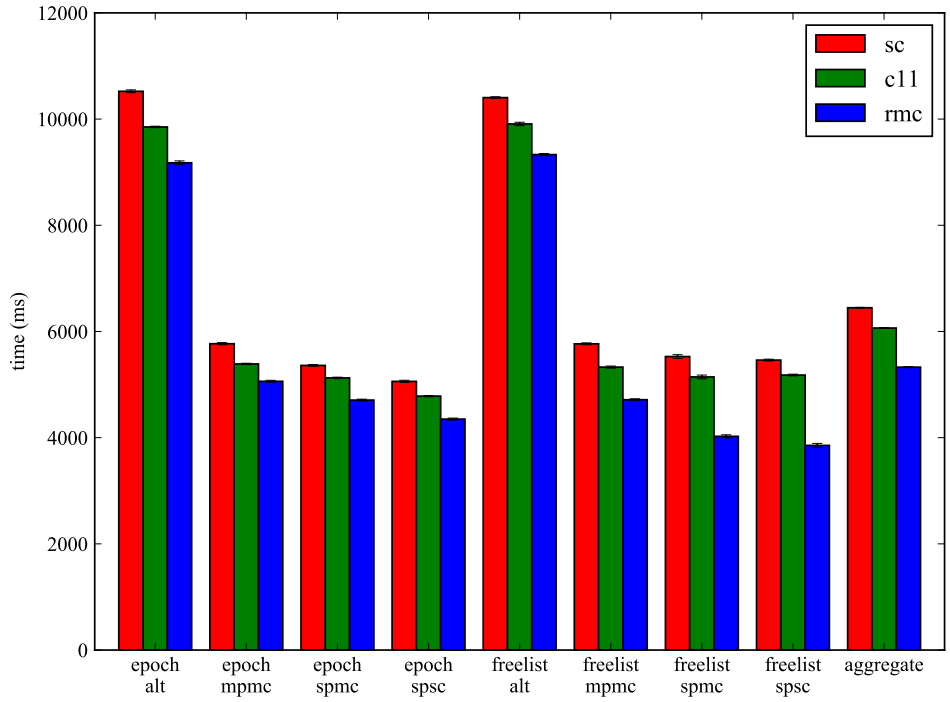
On ARMv8 and Power, RMC suffers in some of the most highly contended tests. On ARMv8, I suspect this is because RMC may not use a "Release" to implement a visibility edge to a compare and swap action (for reasons discussed in Section 4.5.2). Instead of using a release write as part of the compare and swap, then, a `dmb` is executed before the CAS, on each attempt. Power does not have release writes instructions, but there is a similar dynamic at play: C++11 release CAS actions can be compiled so that the `lwsync` is only executed if the read value matches the expected one—RMC does not permit this compilation, so we again need to execute a barrier before each CAS attempt.

Epoch library

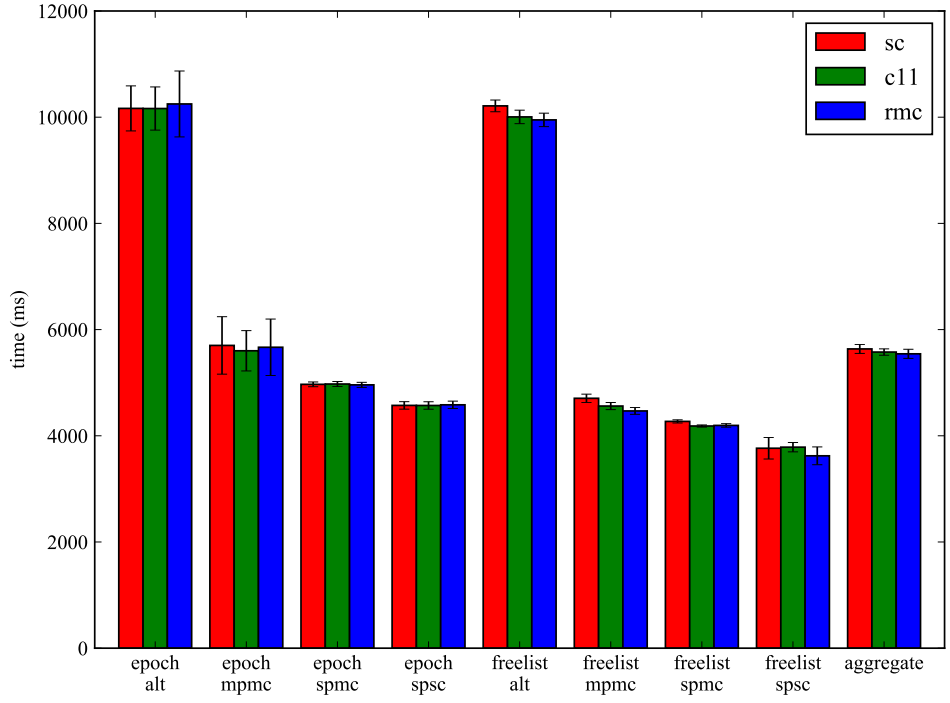
In Figures 6.3 and 6.4, we investigate the performance of different implementations of the epoch memory reclamation library while using it for queues, stacks, and RCU protected lists. For each data structure, we hold the implementation of the structure constant while varying the library: stacks and queues use the "c11" version while RCU-protected lists use the "linux" version. Unsurprisingly, it turns out not to matter much: both tested epoch implementations that use weak orderings perform at about the same level for most workloads; they all generate essentially the same code for the small fast-path code and any differences in the more complex slow-path are drowned out by its infrequency. RMC manages a slight win on ARMv8, I believe due to judicious use of the `dmb st; dmb ld` optimization. On ARMv7, the SC version, which has a noticeably worse fast-path, suffers heavily. On ARMv8, which features excellent SC atomics, the gap between the SC version and the others is much smaller.

RCU list search

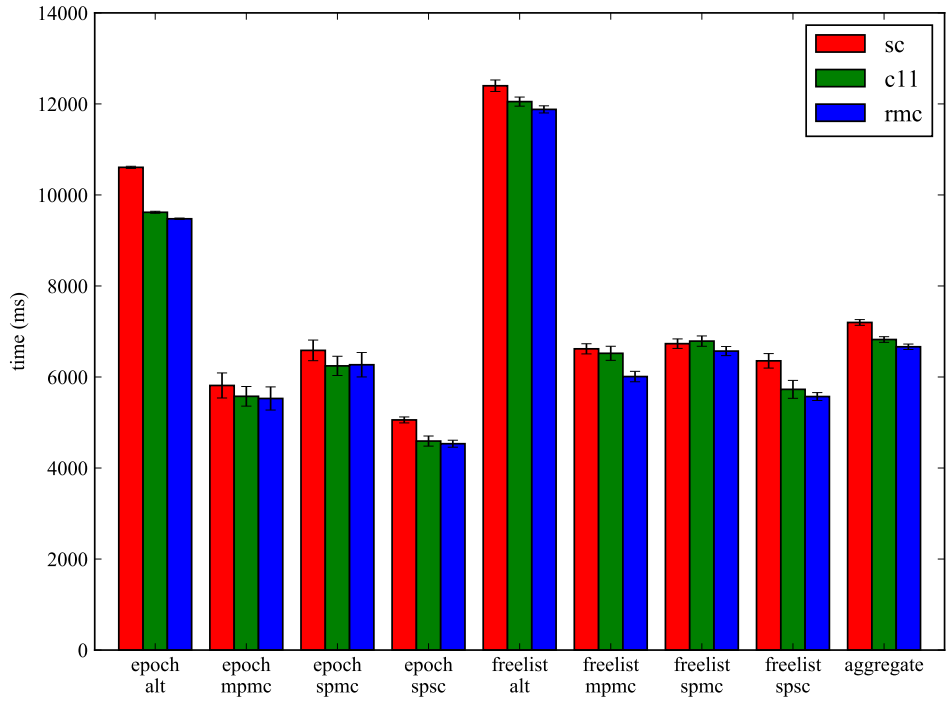
In Figure 6.5 we show the results of testing operations on a list protected by RCU (as discussed in Section 5.7). Since RCU is optimized for read-mostly data structures, the test interleaves lookups of list elements and list modifications with one modification every N lookups (where $N = 10000$ in the regular test and $N = 30$ in the "write-heavy" test). As discussed earlier, list modifications may occur while readers are traversing the list, so reader threads must take care to



(a) ARMv7 results

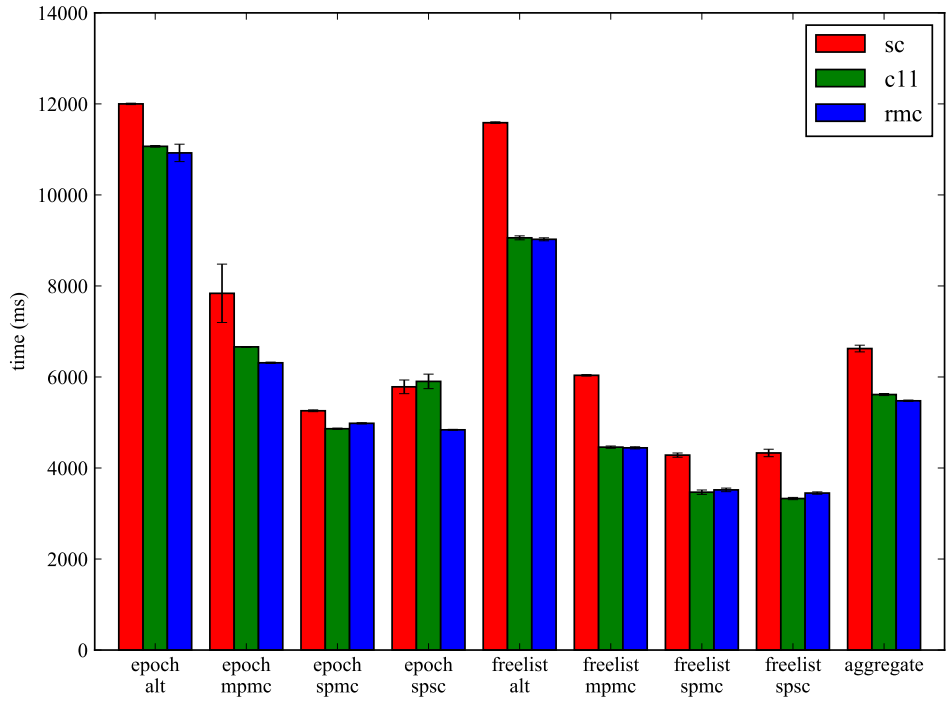


(b) ARMv8 results

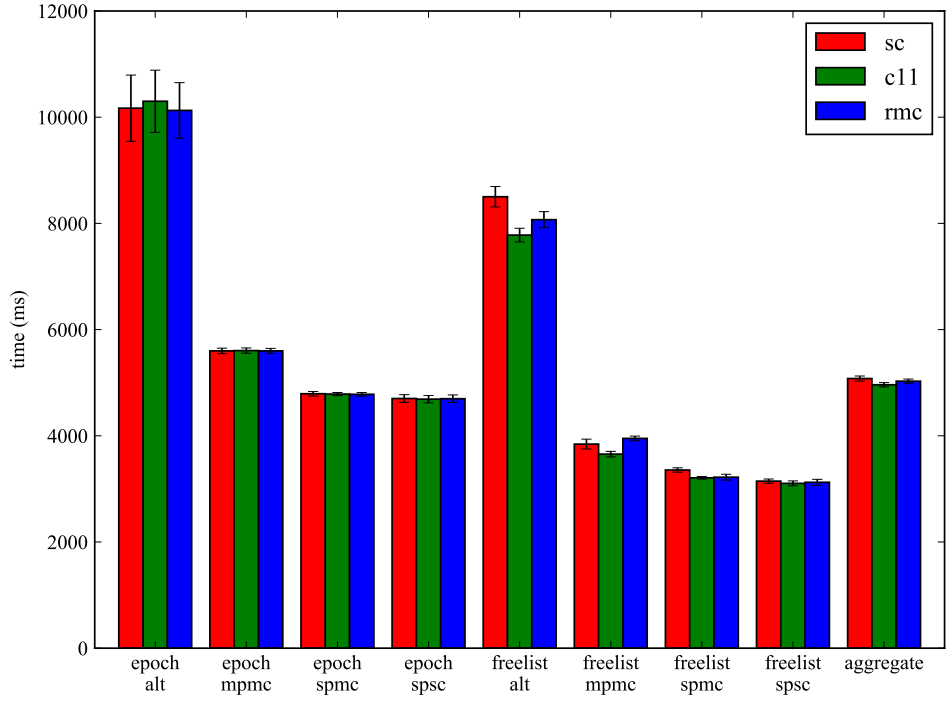


(c) Power results

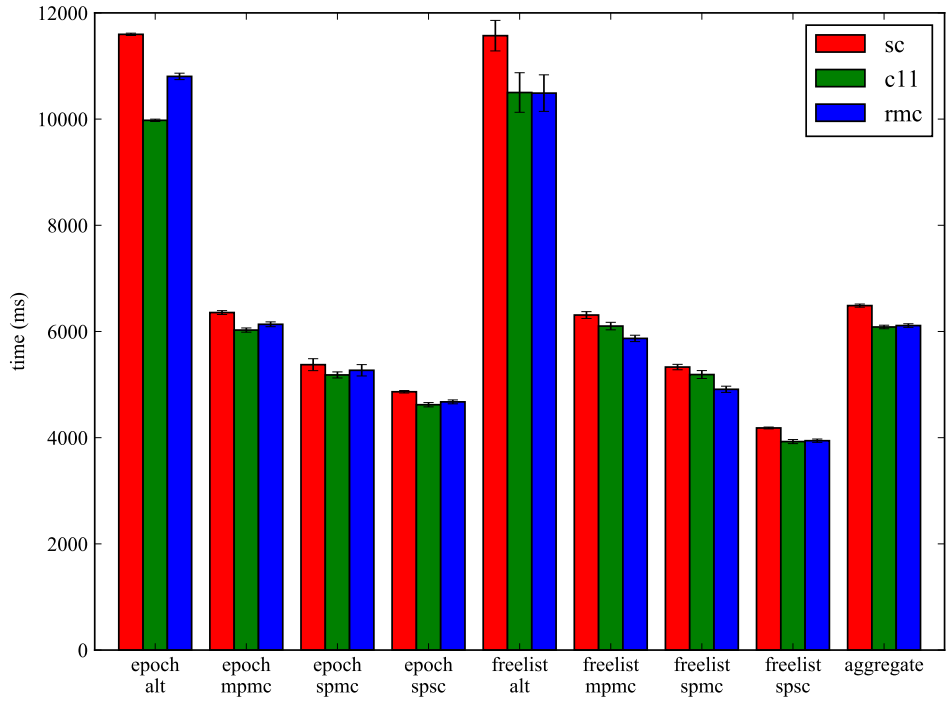
Figure 6.1: Concurrent queue benchmark



(a) ARMv7 results

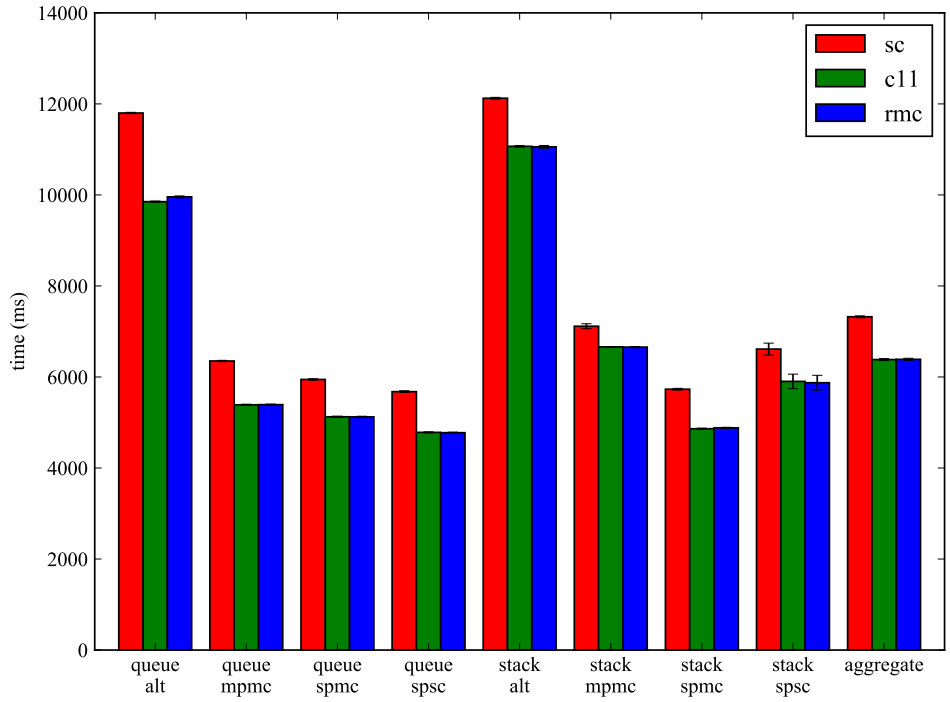


(b) ARMv8 results

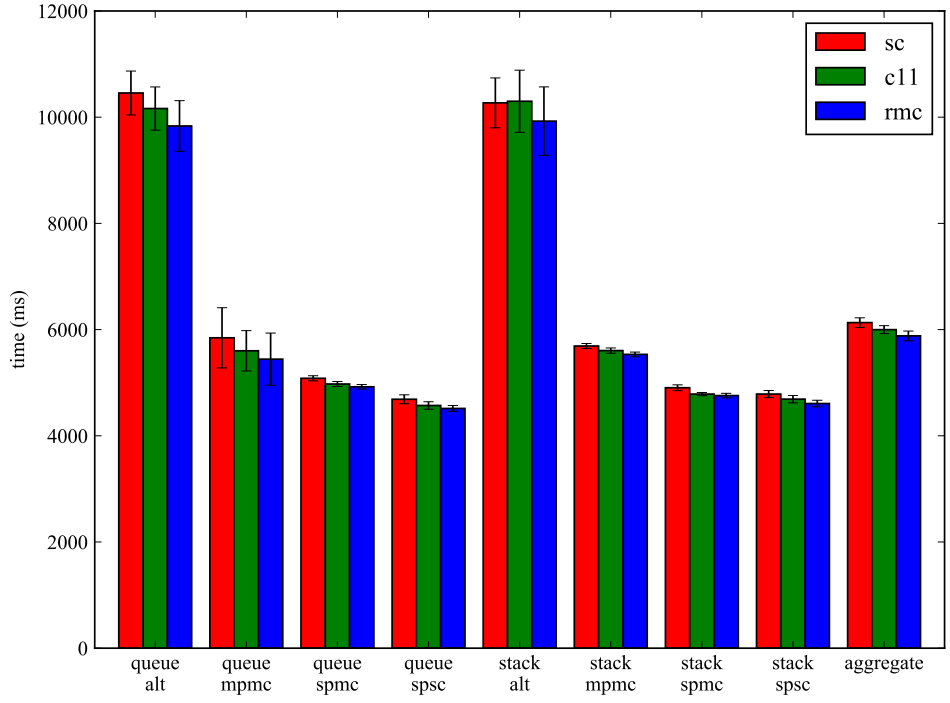


(c) Power results

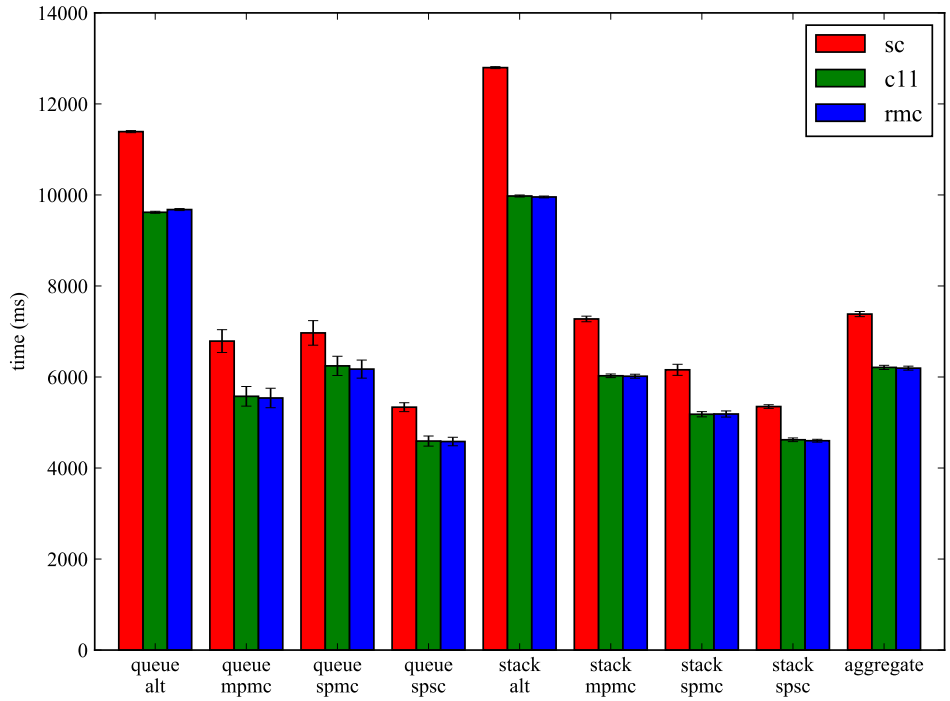
Figure 6.2: Concurrent stack benchmark



(a) ARMv7 results

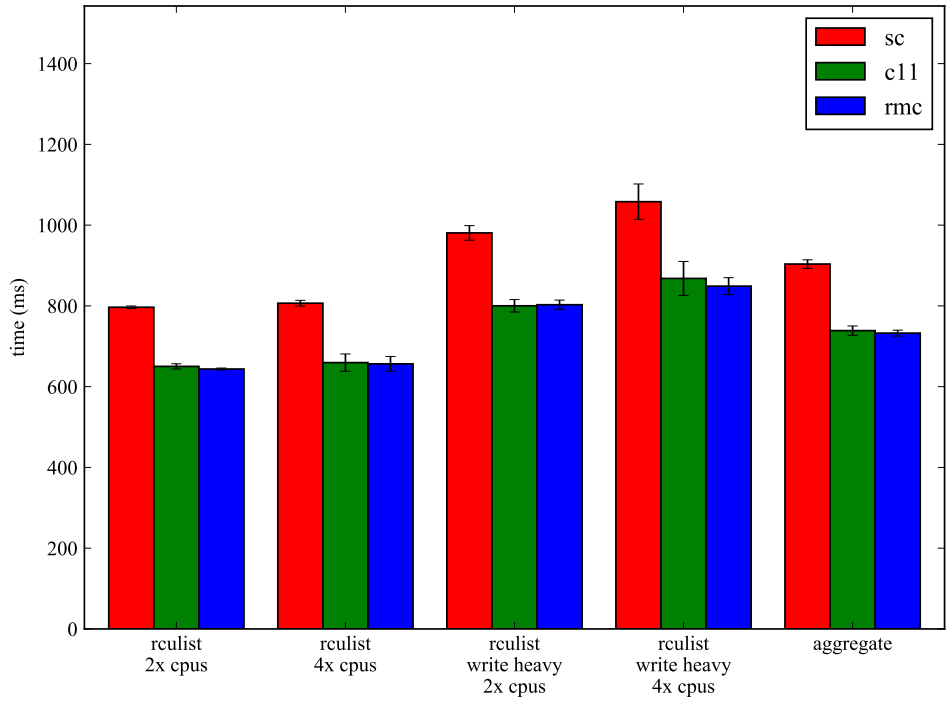


(b) ARMv8 results

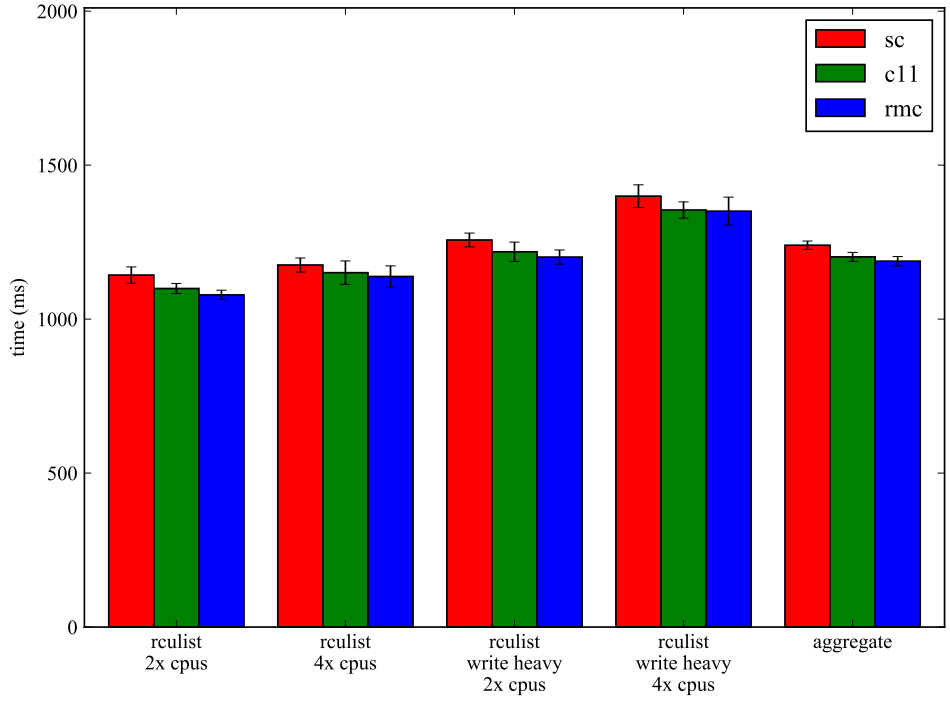


(c) Power results

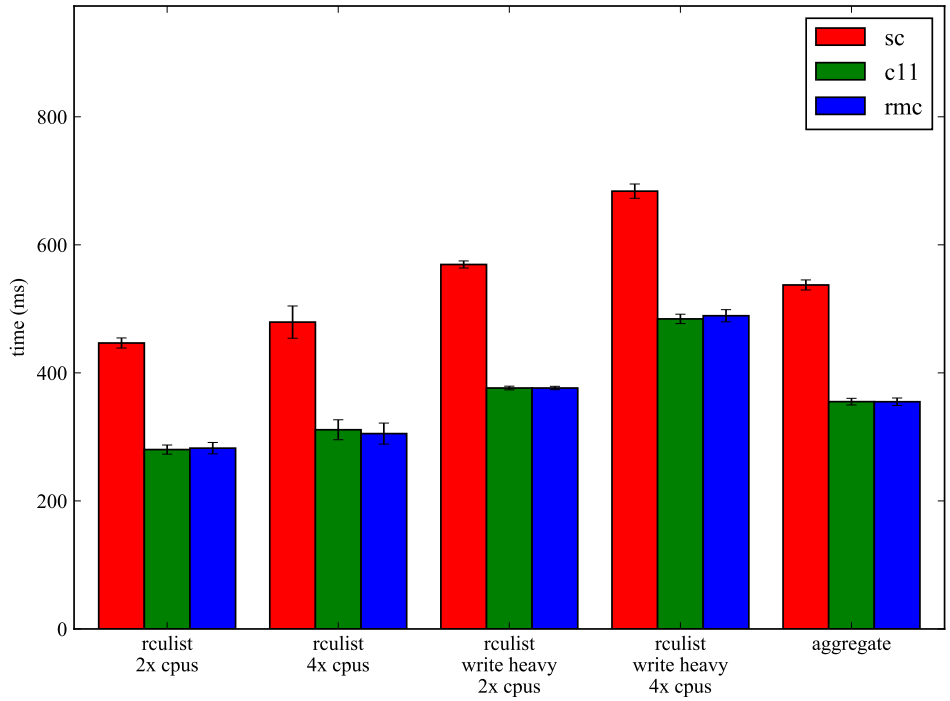
Figure 6.3: Epoch library benchmark



(a) ARMv7 results



(b) ARMv8 results



(c) Power results

Figure 6.4: Epoch library RCU benchmark

ensure that they actually see the contents of any node they get a pointer to. This turns out to be a perfect case for taking advantage of existing data dependencies to enforce ordering.

Here, the C++11 version uses `memory_order_consume`, which establishes ordering based on data dependencies. Unfortunately, consume remains not-implemented-as-intended on all major compilers, which simply emit barriers, leading to uninspiring performance. The RMC version uses fine-grained execution orderings established using `XEDGE_HERE` and successfully avoids emitting any barriers on the read side. The "linux" test is the C++11 version with all uses of consume replaced with relaxed memory accesses; this, following the standard practice in the Linux kernel, confronts the lack of assurances from the language by instead attempting to quantify over all possible compiler transformations [33]. While this approach is fraught with danger, it has served extremely well from a performance perspective.

On ARMv7 and Power, the differences are astounding. RCU is the killer app for taking advantage of data dependencies to enforce ordering semantics, and the over 4x speedup that we get on ARMv7 (and nearly 2x and Power) is a pretty clear justification for all of the pain people have endured in an attempt to take advantage of it. Being able to match the performance of the dangerous Linux-style of RCU use while providing a well-defined semantics is one of the major successes of RMC and `rmc-compiler`.

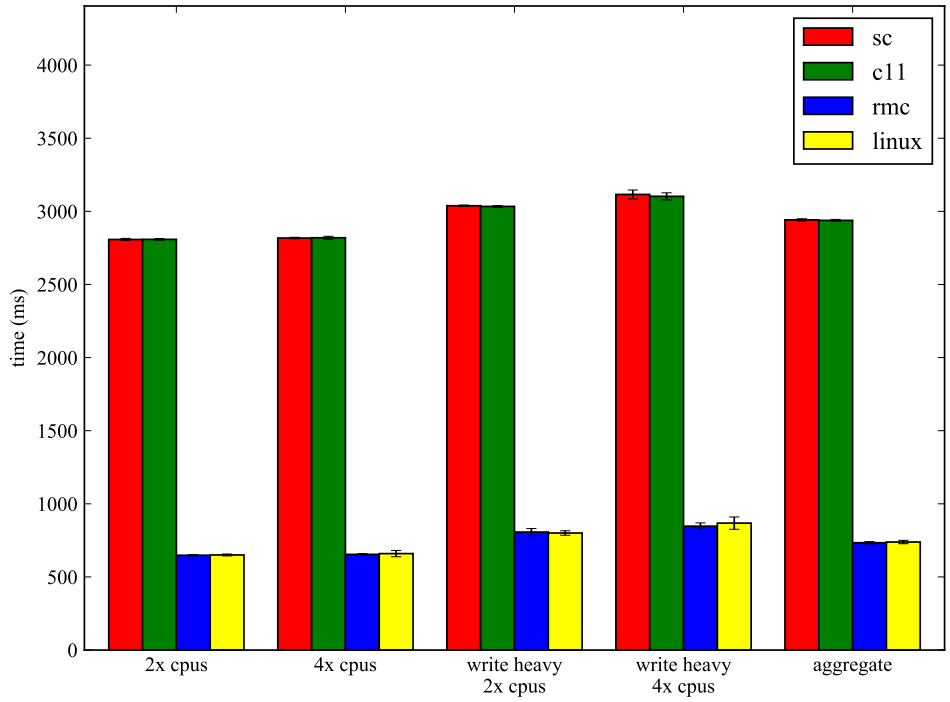
On ARMv8—or at least, on the particular Cortex-A53 being used for testing—it all seems pretty pointless. Using Load-Acquire instructions to enforce ordering while traversing a linked list is as efficient as taking advantage of the data dependencies is. This is impressive, and we wonder how general it is. In RCU list search, essentially all of the memory accesses during the list search are data-dependent loads from the list nodes. Load-Acquire establishes ordering with all subsequent accesses, but in this case that is probably not all that much stronger. The situation may be different on a workload where there is more of a mixture of data-dependent loads with ordering requirements and other loads, and for that reason we continue to keep data dependency use enabled on ARMv8 despite its troubles here. Still, though, it seems likely that the list search case is the more common one. Frustratingly, the RMC version slightly *underperforms* the C++11 and Linux versions. This appears to be due to the techniques for ensuring dependencies are preserved interfering with optimizations. This is almost certainly an artifact of our implementation (which makes no changes to LLVM) and not anything fundamental—tighter integration with a backend should make it possible to be more discriminating in which optimizations are inhibited.

qspinlocks

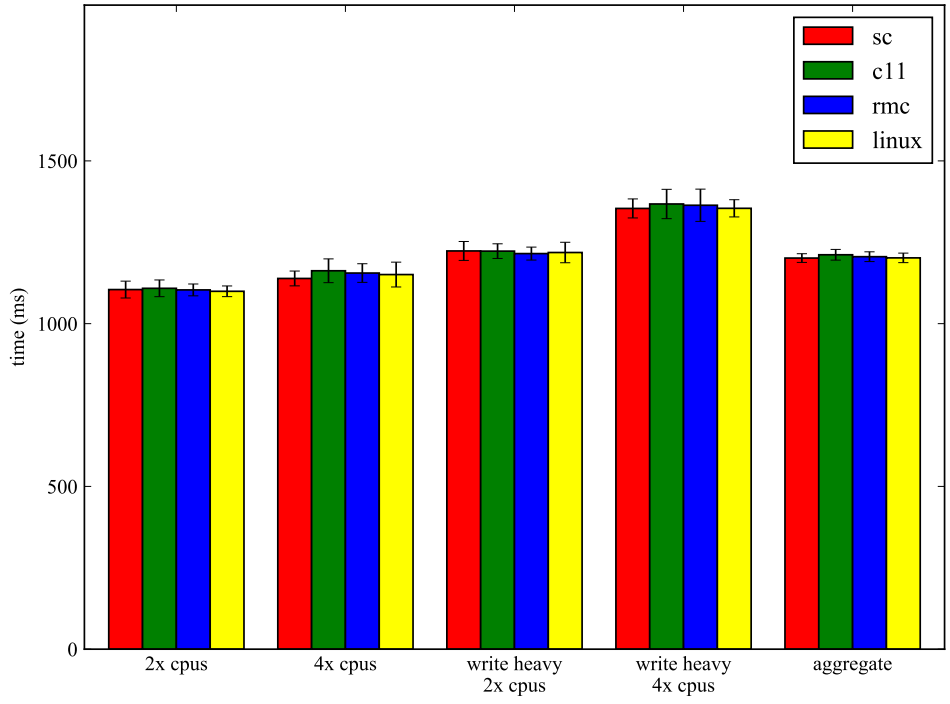
The last benchmark we consider are qspinlocks, which we discussed in Section 5.5. We show qspinlock performance results in Figure 6.6. The workload in the benchmark here is an extremely heavily contended one in which mutator threads attempt to reacquire the lock nearly immediately after relinquishing it. This means that except for the 1 CPU test, which (tautologically) exclusively tests the uncontended fast-path, it is exclusively the slow-path being tested.

The ARMv7 benchmark results here are pretty fascinating. RMC wins handily on 3 or 4 CPUs but loses with 2 CPUs. This is a somewhat fascinating tradeoff, though we are not sure whether it is a fundamental one or whether there exists a variant that would win for all processor counts.

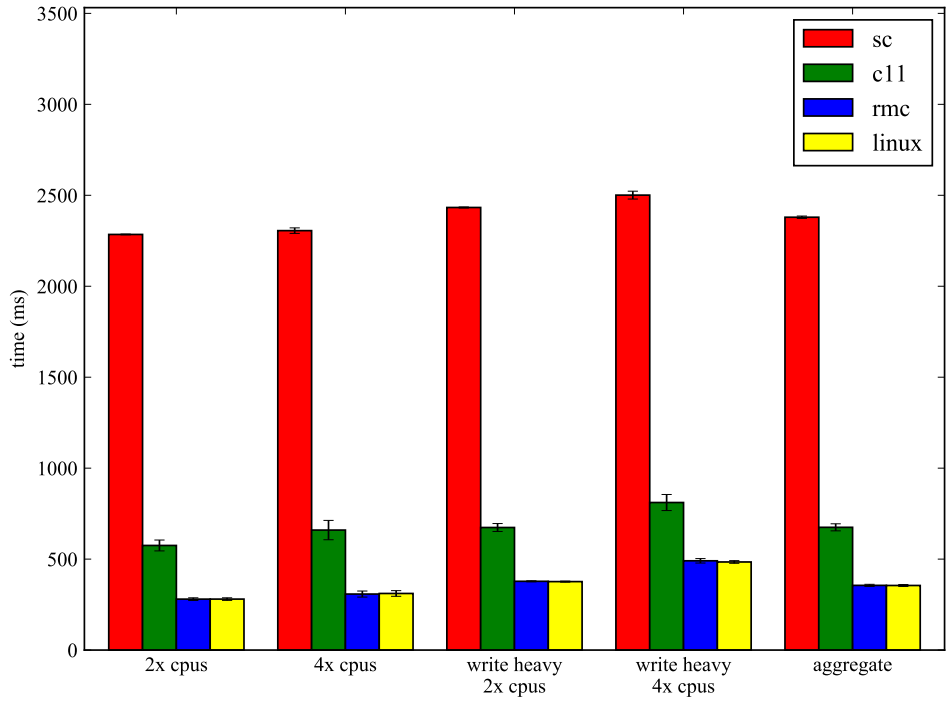
ARMv8 has similar but different and smaller effects: winning slightly on 2 and 4 CPUs and



(a) ARMv7 results



(b) ARMv8 results



(c) Power results

Figure 6.5: RCU list search benchmark

losing on 3. I am not really sure what to make of this.

Power gives nice consistent wins.

Ring buffers

The last benchmark we consider is a single-producer/single-consumer ring buffer similar to that in 2.2.1, but generalized to holding arbitrary types (and thus requires pre and post edges to provide reasonable semantics to callers). We show performance results in Figure 6.7. Because the buffers are single-producer/single-consumer, we only have one test configuration per implementation.

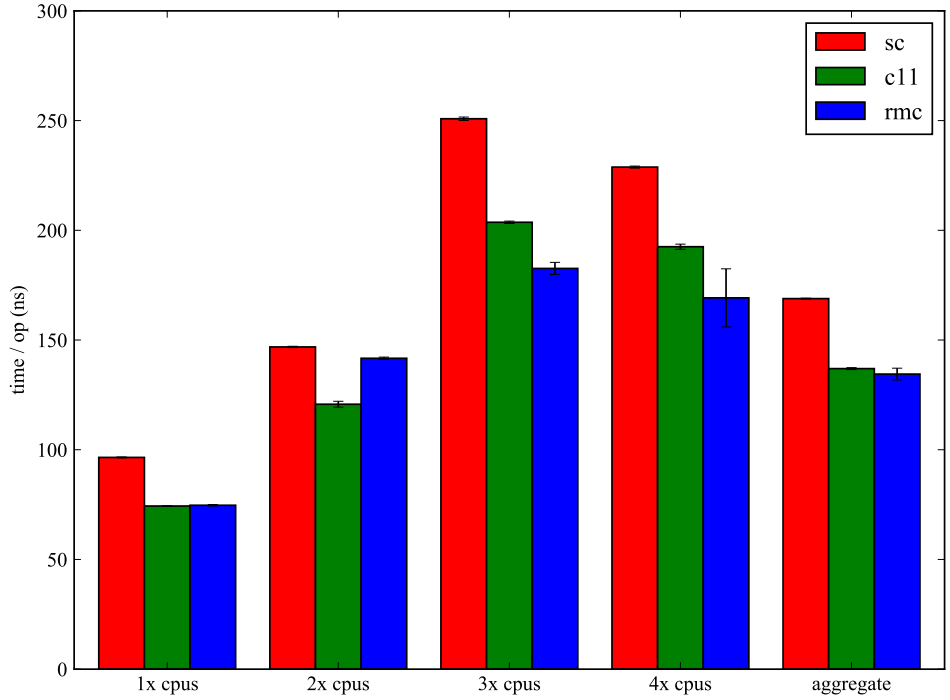
RMC performs well here on every tested architecture, as it is able to take advantage of control dependencies to writes that C++11 is not.

6.2 Compiler performance

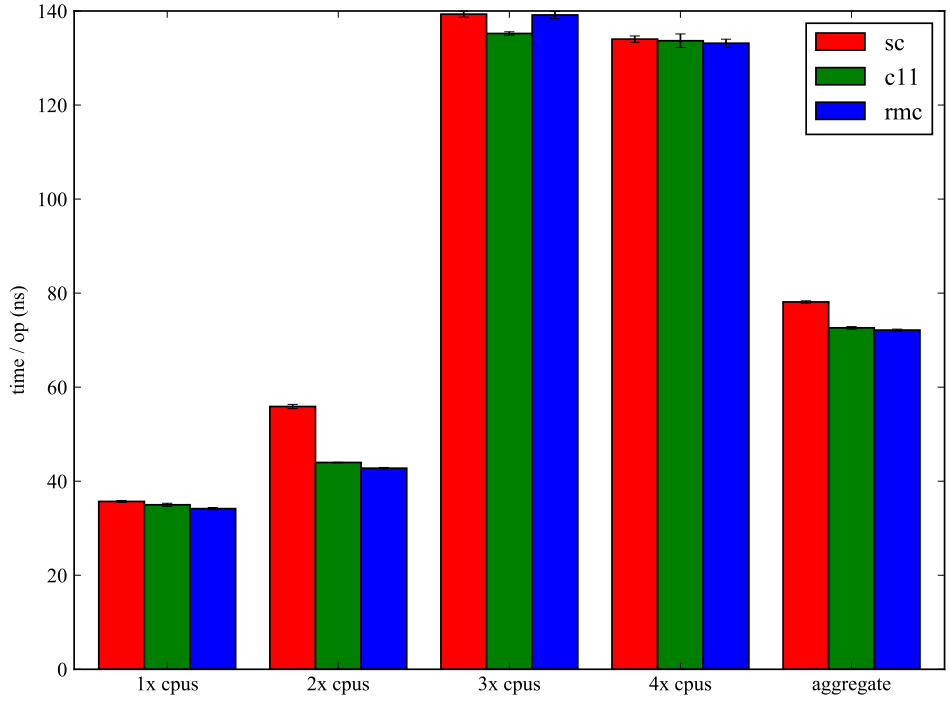
Given the rather high asymptotic complexity of our SMT-based algorithms (the size of the generated SMT problems is linear in the number of paths through the control flow graph between the sources and destinations of edges, which can be exponential in the worst case), it is natural to wonder about the performance of the compiler itself.

Compile times for a number of programs is given in Figure 6.8. Compile time on ARMv7 is consistently good while compile time on ARMv8 and POWER, which have more options for how to compile constraints—and thus a larger SMT state space—can get nasty. There are a number of approaches that could be taken to improve compilation time—attempting to reduce the SMT state space by eliminating unlikely options, adding RMC-tuned heuristics to Z3, specifying a timeout to Z3 and choosing the best found, or designing a real algorithm for RMC compilation (likely an approximation algorithm).

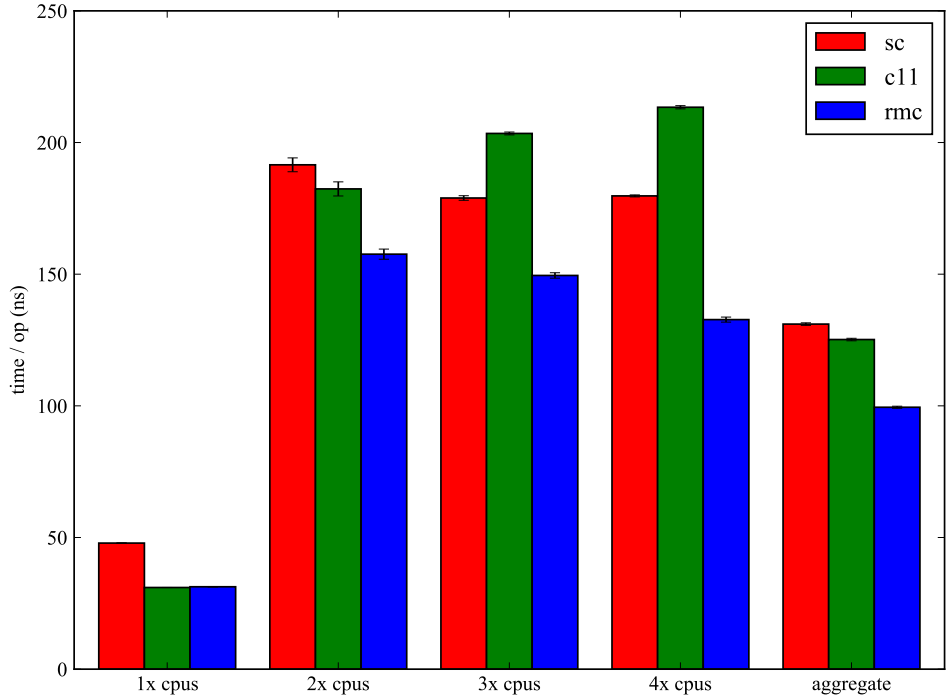
We are not terribly worried about our compilation time, however, as the sort of low-level concurrent code that uses RMC is likely to be a relatively small portion of any real codebase. The one seeming potential exception to that is likely to be in handling RCU-protected data, which requires execution edges to all use sites. That is a relatively simple use of RMC, that can easily be compiled well without resort to an SMT solver. If compilation time for large RCU-protected code paths proved a problem, then, one simple solution would be heuristically falling back to our non-SMT greedy compilation algorithm.



(a) ARMv7 results



(b) ARMv8 results



(c) Power results

Figure 6.6: qspinlock

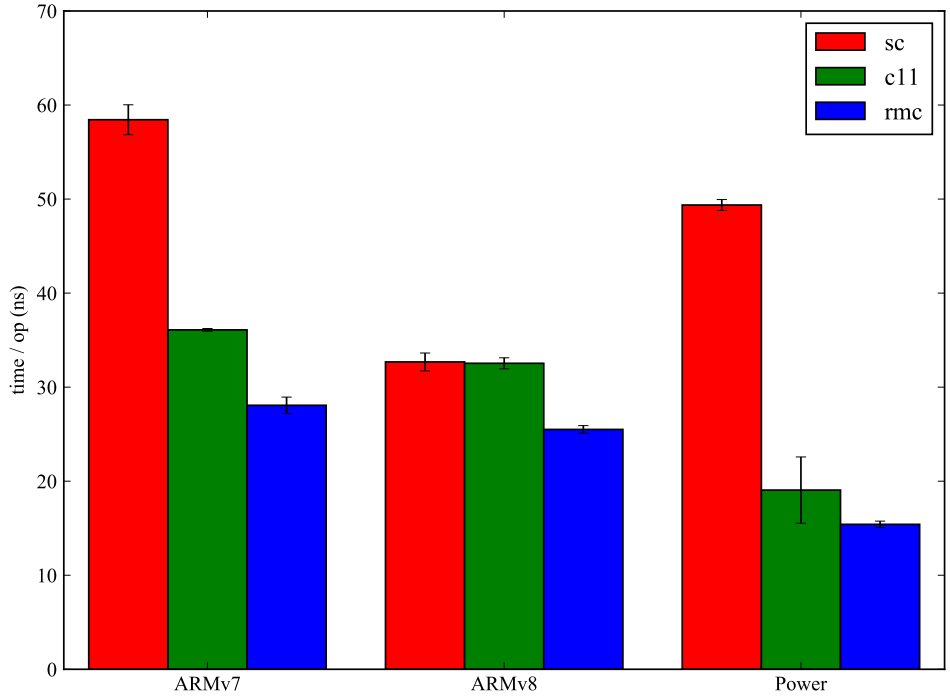


Figure 6.7: Ring buffers

Test	C11 time	RMC time	Slowdown
epoch	1.61	1.82	1.13x
ms_queue epoch	2.54	2.66	1.05x
ms_queue freelist	2.49	2.60	1.05x
tstack epoch	2.49	2.56	1.03x
tstack freelist	2.46	2.53	1.03x
rculist	2.25	2.28	1.01x
ringbuf	2.08	2.16	1.04x
qspinlock	2.32	2.63	1.14x
seqlock	2.26	2.29	1.02x
aggregate			1.05x

(a) ARMv7 results

Test	C11 time	RMC time	Slowdown
epoch	3.55	7.28	2.05x
ms_queue epoch	5.90	7.83	1.33x
ms_queue freelist	5.82	11.34	1.95x
tstack epoch	5.79	6.23	1.07x
tstack freelist	5.73	6.15	1.07x
rculist	5.22	5.32	1.02x
ringbuf	4.88	5.20	1.07x
qspinlock	5.43	31.00	5.71x
seqlock	5.24	5.37	1.03x
aggregate			1.50x

(b) ARMv8 results

Test	C11 time	RMC time	Slowdown
epoch	0.60	1.69	2.83x
ms_queue epoch	1.00	2.45	2.46x
ms_queue freelist	0.98	1.15	1.17x
tstack epoch	0.92	1.37	1.49x
tstack freelist	0.91	1.13	1.25x
rculist	0.82	0.84	1.03x
ringbuf	0.76	0.85	1.12x
qspinlock	0.85	4.40	5.19x
seqlock	0.82	0.84	1.03x
aggregate			1.65x

(c) POWER results

Figure 6.8: Compilation times

Chapter 7

Conclusion

7.1 Usability

For most of the algorithms and data structures I constructed, I found the RMC versions easier to understand, reason about, and modify.

Because relative ordering is such a fundamental concept for low-level weak memory programming, writing programs in a traditional style still requires careful thought about relative ordering, which must then be mapped by the programmer onto the concrete primitives. By making ordering constraints explicit, we pass this responsibility off to the compiler. On top of that, modifying programs written in a traditional style will typically involve needing to reconstruct the necessary ordering properties in order to ensure that changes do not violate any of the required orderings. Because fences and ordering annotations are relatively coarse-grained tools, the existing fences and orderings will typically not be enough to precisely reconstruct these constraints—they must be supplemented by reference back to the algorithm's design, comments, tea leaves, etc to recover this understanding. This must be done every time modifications are made. These constraints can of course be documented, but as always that is likely to drift from the code. Much easier is to take advantage of the ordering constraints directly.

RMC fares well when compared against C++11-style memory order annotations, but it *especially* shines when compared to using explicit memory fences. The above problems occur when using C++11-style memory order annotations but are particularly salient when using explicit fences, where a single fence is often used to resolve many constraints. Furthermore, the definition of C++11's fences is probably the hairiest part of the system. The core C++11 notion of `synchronizes-with` must be defined in cases for all possible combinations of fences and memory order annotations that may induce it. Sequentially consistent fences are in particular quite troublesome; not only do they have particularly complex and hard to reason about behavior, they are also quite weak: it is impossible to use C++11 SC fences to recover sequential consistency. RMC pushes serve as a better and easier to reason about alternative to C++11 sequentially consistent fences, while visibility constraints take the role of release and acquire/release fences and execution constraints take the role of acquire fences.

RMC gets enormous mileage out of just two core concepts: execution and visibility order. Even pushes are defined in terms of them: a push is simply an action that is globally visible as

soon as it executes. This unity of expression allows very similar reasoning to be used for all sorts of programs, from simple message passing to subtle constructs such as sequence locks (which seem much less subtle in RMC than they do in C++11!).

While I obviously come from a biased perspective, in many cases I found that the easiest way to write C++11 atomics code was to—in essence—run the RMC compilation algorithm in my head. This was especially true in cases where I wished to use explicit fences.

7.2 Performance

The performance picture for compiling RMC with `rmc-compiler` is quite encouraging. In our ARMv7 tests, RMC achieved consistent and noticeable wins compared to C++11. For RCU protected linked-list search, we achieve enormous speedups while still providing a well-defined semantics.

In our ARMv8 tests, things are less exciting—we eke out a number of very marginal improvements over C++11, along with a very marginal loss on RCU list search (which is our pride and joy on ARMv7!). This is disappointing (who doesn't like to win?), but it is neither surprising nor particularly damaging to the case for RMC. ARMv8 essentially built the C++11 memory model into hardware—of course it performs well! One wonders what we could do with hardware support designed for RMC itself! If we believe that RMC is a better programmer model—and we do, as argued above—parity in performance is more than adequate. While the gap with SC atomics is much narrower in our ARMv8 tests than in our ARMv7 tests, it is still present, and so using weaker atomics will still be worthwhile in some cases—even discounting portable code that will also need to be run on architectures with a wider gap between sequentially consistent atomics and weaker ones.

7.3 Related work

Using integer linear programming (ILP) to optimize the placement of barriers is a common technique. Bouajjani et al. [14] and Alglave et al. [1] both use ILP to calculate where to insert barriers to recover sequential consistency as part of tools for that purpose. Bender et al. [8] use ILP to place barriers in order to compile what is essentially a much simplified version of RMC (with only one sort of edge, most akin to RMC's push edges). We believe we are the first to extend this technique to handle the use of dependencies for ordering.

Ou and Demsky's AutoMO [39] attempts to simplify the use of the C++11 model by automatically inferring memory orders: given a data structure implementation and a test suite, AutoMO finds a assignment of memory orders such that the data structure exhibits only SC behaviors under the test suite.

OpenMP's [9] flush directive also allows a form of pairwise ordering, but the programming model has little in common with RMC: OpenMP flush is just a barrier—with restrictions on what locations it applies to—and the pairwise ordering is based on locations and not specific actions.

The C++ concurrency working group is working to repair the `memory_order_consume` implementability problem, though nothing seems final yet [35, 36].

7.4 Future work

Despite our generally good performance, there is much work that could be done to improve `rmc-compiler`. While `rmc-compiler` is limited by being an LLVM pass that works with an otherwise unmodified Clang and LLVM, there are other improvements that could be made in a hypothetical RMC compiler that is more tightly integrated with the rest of the compiler. The code modifications made to ensure the preservation of data dependencies hide information from the optimizer—this prevents the optimizer from causing trouble but also can result in worse code being generated. Likewise, barriers are inserted to enforce ordering, but this overconstrains later passes by preventing *all* memory accesses from being moved across the barrier—better would be to be able to insert barriers with annotations on *which* operations are not permitted to be moved across it.

While it seems to work well in many cases, the entire optimization approach based on assigning a fixed weight to each type of operation and scaling it based on how often we expect it to execute is faulty. Performance characteristics of modern processors are much more complicated than that.

The SMT-based algorithm for our compilation optimization seems to work well, but its general algorithmic approach is best characterized as "brute force and ignorance." Perhaps an algorithm that actually involved some design could achieve similar performance characteristics in polynomial time.

There are a number of features of the C++11 concurrency model that were left out of RMC in order to achieve a simpler and more elegant model. It may—or may not—be worth adding some of them in for performance reasons. C++11 has a notion of "release sequences", which, roughly, allows a thread that has performed a release write to perform relaxed writes to the same location without destroying the message-passing behavior of the release. This is a somewhat niche use case, but it comes up in the implementation of epochs—we generate worse code so as to not rely on behavior that RMC does not promise. Unfortunately the definition of release sequences is quite hairy in C++11 and it is not at all obvious how it would be ported to RMC.

A less awful feature would be allowing the drawing of finer ordering distinctions based on whether a compare-and-swap succeeded or failed. C++11 supports this, and there are cases where it may generate better code as a result. Having different constraints *out* of a compare-and-swap based on whether it succeeded can in principle already be done using control flow, but RMC's practice of taking the transitive closure of the constraint graph makes taking advantage of this difficult. Being able to draw a visibility edge *to* a compare-and-swap when it succeeds but not when it fails would help RMC performance in a number of cases, especially on ARMv8. Giving semantics to this seems eminently doable, though likely to be inelegant.

While RMC has a rigorously defined core calculus, which should aid reasoning about the correctness of RMC programs, it lacks a program logic for doing this in a rigorous and formal way. Some preliminary work on designing a concurrent separation logic for RMC was done by Joseph Tassarotti and Karl Crary (the RMC 2.0 rework was done to make RMC more amenable to such), but it has not been completed.

As part of their C++11 formalization work [6], Batty et al. created CPPMEM, a tool for exploring the potential outcomes of a concurrent C++ program. While it is fairly limited and not suitable for verifying full algorithms, it is incredibly useful for examining small test cases and

building an intuition for the model. An equivalent tool for RMC would be an excellent help for developers.

Appendix A

Full RMC recap

A.1 Syntax

types	τ	::=	$\mathsf{unit} \mid \mathsf{nat} \mid \tau\ \mathsf{ref} \mid \tau\ \mathsf{susp} \mid \tau \to \tau$
numbers	n	::=	$0 \mid 1 \mid \cdots$
tags	T	::=	$\cdots$
locations	ℓ	::=	$\cdots$
threads	p	::=	$\cdots$
terms	m	::=	$x \mid \texttt{()} \mid \ell \mid n \mid \mathsf{ifz}\ m\ \mathsf{then}\ m\ \mathsf{else}\ m$
		$\mid$	$\mathsf{susp}\ e \mid \mathsf{fun}\ x(x{:}\tau){:}\tau.m \mid m\ m$
values	v	::=	$x \mid \texttt{()} \mid \ell \mid n \mid \mathsf{susp}\ e \mid \mathsf{fun}\ x(x{:}\tau){:}\tau.m$
attributes	b	::=	$\mathsf{vis} \mid \mathsf{exe} \mid$
labels	φ	::=	$t \mid T \mid \overset{b}{\triangleleft} \mid \overset{b}{\triangleright}$
action type	t	::=	$\mathsf{c} \mid \mathsf{sc} \mid \mathsf{p}$
expr's	e	::=	$\mathsf{ret}\ m \mid x \leftarrow e\ \mathsf{in}\ e$
		$\mid$	$\mathsf{force}\ m$
		$\mid$	$\mathsf{R}_t[m] \mid \mathsf{W}_t[m,m] \mid \mathsf{RMW}_t[m, x{:}\tau.m, x{:}\tau.m]$
		$\mid$	$\mathsf{RMWS}_t[\ell, v, m, m]$
		$\mid$	$\mathsf{Push} \mid \mathsf{Nop}$
		$\mid$	$\mathsf{new}\ t \overset{b}{\Rightarrow} t.e \mid \varphi \,\#\, e$
		$\mid$	$\mathsf{fork}\ e$
		$\mid$	$\mathsf{alloc}\ m{:}\tau$
execution states	ξ	::=	$\epsilon \mid \xi \parallel p : e$
tag sig	Υ	::=	$\epsilon \mid \Upsilon, T$
loc'n sig	Φ	::=	$\epsilon \mid \Phi, \ell{:}\tau$
contexts	Γ	::=	$\epsilon \mid \Gamma, t{:}\mathsf{tag} \mid \Gamma, x{:}\tau$

Figure A.1: RMC Syntax

A.2 Thread static semantics

$\boxed{\Upsilon; \Phi; \Gamma \vdash m : \tau}$

$$\frac{\Gamma(x) = \tau}{\Upsilon; \Phi; \Gamma \vdash x : \tau} \qquad \frac{}{\Upsilon; \Phi; \Gamma \vdash \texttt{()} : \mathsf{unit}} \qquad \frac{\Phi(\ell) = \tau}{\Upsilon; \Phi; \Gamma \vdash \ell : \tau\ \mathsf{ref}}$$

$$\frac{\Upsilon; \Phi;\ \Gamma \vdash e : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{susp}\, e : \tau\ \mathsf{susp}} \qquad \frac{}{\Upsilon; \Phi; \Gamma \vdash n : \mathsf{nat}}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \mathsf{nat} \quad \Upsilon; \Phi; \Gamma \vdash m_2 : \tau \quad \Upsilon; \Phi; \Gamma \vdash m_3 : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{ifz}\, m_1\, \mathsf{then}\, m_2\, \mathsf{else}\, m_3 : \tau}$$

$$\frac{\Upsilon; \Phi; (\Gamma, f{:}(\tau_1 \to \tau_2), x{:}\tau_1) \vdash m : \tau_2 \quad f, x \notin \mathsf{Dom}(\Gamma)}{\Upsilon; \Phi; \Gamma \vdash \mathsf{fun}\ f(x{:}\tau_1){:}\tau_2.m : \tau_1 \to \tau_2}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \tau \to \tau' \quad \Upsilon; \Phi; \Gamma \vdash m_2 : \tau}{\Upsilon; \Phi; \Gamma \vdash m_1 m_2 : \tau'}$$

$\boxed{\Upsilon; \Gamma \vdash \varphi : \mathsf{label}}$

$$\frac{t{:}\mathsf{tag} \in \Gamma}{\Upsilon; \Gamma \vdash t : \mathsf{label}} \qquad \frac{T \in \Upsilon}{\Upsilon; \Gamma \vdash T : \mathsf{label}}$$

$$\frac{}{\Upsilon; \Gamma \vdash \overset{b}{\triangleleft} : \mathsf{label}} \qquad \frac{}{\Upsilon; \Gamma \vdash \overset{b}{\triangleright} : \mathsf{label}}$$

$\boxed{\Upsilon; \Phi; \Gamma \vdash e : \tau}$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{ret}\, m : \tau} \qquad \frac{\Upsilon; \Phi; \Gamma \vdash m : \tau\ \mathsf{susp}}{\Upsilon; \Phi; \Gamma \vdash \mathsf{force}\, m : \tau}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash e_1 : \tau_1 \quad \Upsilon; \Phi; (\Gamma, x{:}\tau_1) \vdash e_2 : \tau_2 \quad x \notin \mathsf{Dom}(\Gamma)}{\Upsilon; \Phi; \Gamma \vdash x \leftarrow e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m : \tau\ \mathsf{ref}}{\Upsilon; \Phi; \Gamma \vdash \mathsf{R}[m] : \tau} \qquad \frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \tau\ \mathsf{ref} \quad \Upsilon; \Phi; \Gamma \vdash m_2 : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{W}[m_1, m_2] : \mathsf{unit}}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash m_1 : \tau\ \mathsf{ref} \quad \Upsilon; \Phi; \Gamma, x{:}\tau \vdash m_2 : \mathsf{nat} \quad \Upsilon; \Phi; \Gamma, x{:}\tau \vdash m_3 : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{RMW}[m_1, x{:}\tau.m_2, x{:}\tau.m_3] : \tau}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash \ell : \tau\ \mathsf{ref} \quad \Upsilon; \Phi; \vdash v : \tau \quad \Upsilon; \Phi; \vdash m_1 : \tau \quad \Upsilon; \Phi; \vdash m_2 : \mathsf{nat}}{\Upsilon; \Phi; \Gamma \vdash \mathsf{RMWS}[\ell, v, m_1, m_2] : \tau}$$

$$\frac{}{\Upsilon; \Phi; \Gamma \vdash \mathsf{Push} : \mathsf{unit}} \qquad \frac{}{\Upsilon; \Phi; \Gamma \vdash \mathsf{Nop} : \mathsf{unit}}$$

$$\frac{\Upsilon; \Phi; (\Gamma, t{:}\mathsf{tag}, t'{:}\mathsf{tag}) \vdash e : \tau \quad t, t' \notin \mathsf{Dom}(\Gamma)}{\Upsilon; \Phi; \Gamma \vdash \mathsf{new}\ t \overset{b}{\Rightarrow} t'.e : \tau} \qquad \frac{\Upsilon; \Gamma \vdash \varphi : \mathsf{label} \quad \Upsilon; \Phi; \Gamma \vdash e : \tau}{\Upsilon; \Phi; \Gamma \vdash \varphi \# e : \tau}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash e : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{fork}\, e : \mathsf{unit}} \qquad \frac{\Upsilon; \Phi; \Gamma \vdash m : \tau}{\Upsilon; \Phi; \Gamma \vdash \mathsf{alloc}\ m{:}\tau : \tau\ \mathsf{ref}}$$

$\boxed{\Upsilon; \Phi; \vdash \xi\ \mathsf{ok}}$

$$\frac{}{\Upsilon; \Phi \vdash \epsilon\ \mathsf{ok}} \qquad \frac{p \notin \xi \quad \Upsilon; \Phi \vdash \xi\ \mathsf{ok} \quad \Upsilon; \Phi; \epsilon \vdash e : \tau}{\Upsilon; \Phi \vdash \xi \parallel p : e\ \mathsf{ok}}$$

A.3 Thread dynamic semantics

$$\begin{array}{lllll}
\text{identifiers} & i & ::= & \cdots & \\
\text{actions} & \alpha & ::= & \mathsf{R}_t[\ell] \mid \mathsf{W}_t[\ell, v] \mid \mathsf{RW}_t[\ell, v] & \\
& & \mid & \mathsf{Push} \mid \mathsf{Nop} & \\
\text{transactions} & \delta & ::= & \emptyset \mid \vec{\varphi} \# i = \alpha \downarrow v \mid T \overset{b}{\Rightarrow} T \mid m : \tau & \\
& & \mid & \mathsf{fork}\, e\, \mathsf{as}\, p \mid \mathsf{alloc}\ v{:}\tau \mathsf{as}\, \ell &
\end{array}$$

Memory access expressions and actions are subscripted with an action type t. In the interest of simplicity and legibility, we elide the t in situations where it is not interesting. This includes some cases in the dynamic semantics in which the action type is relevant but still not interesting: in those cases, all of the action types in the rule must be the same.

$\boxed{e \overset{\delta}{\mapsto} e'}$

$$\varphi \# \delta = \begin{cases} (\varphi, \vec{\varphi}) \# i = \alpha \downarrow v & \text{if } \delta = (\vec{\varphi} \# i = \alpha =\downarrow v) \\ \delta & \text{otherwise} \end{cases}$$

$$\frac{}{\alpha \overset{\epsilon \# i = \alpha \downarrow v}{\longmapsto} \mathsf{ret}\ v} \qquad \frac{e \overset{\delta}{\mapsto} e'}{\varphi \# e \overset{\varphi \# \delta}{\longmapsto} \varphi \# e'} \qquad \frac{}{\varphi \# \mathsf{ret}\ v \overset{\emptyset}{\mapsto} \mathsf{ret}\ v}$$

$$\frac{}{\mathsf{new}\ t \overset{b}{\Rightarrow} t'.e \overset{T \overset{b}{\Rightarrow} T'}{\longmapsto} [T, T'/t, t']e}$$

$$\frac{e_1 \overset{\delta}{\mapsto} e_1'}{(x \leftarrow e_1\ \mathsf{in}\ e_2) \overset{\delta}{\mapsto} (x \leftarrow e_1'\ \mathsf{in}\ e_2)} \qquad \frac{}{(x \leftarrow \mathsf{ret}\ v\ \mathsf{in}\ e) \overset{\emptyset}{\mapsto} [v/x]e}$$

$$\frac{m \mapsto m'}{\mathsf{ret}\, m \overset{\emptyset}{\mapsto} \mathsf{ret}\, m'}$$

$$\frac{m \mapsto m'}{\mathsf{force}\, m \overset{\emptyset}{\mapsto} \mathsf{force}\, m'} \qquad \frac{}{\mathsf{force}(\mathsf{susp}\, e) \overset{\emptyset}{\mapsto} e}$$

$$\frac{m \mapsto m'}{\mathsf{R}[m] \overset{\emptyset}{\mapsto} \mathsf{R}[m']}$$

$$\frac{m_1 \mapsto m_1'}{\mathsf{W}[m_1, m_2] \overset{\emptyset}{\mapsto} \mathsf{W}[m_1', m_2]} \qquad \frac{m_2 \mapsto m_2'}{\mathsf{W}[v_1, m_2] \overset{\emptyset}{\mapsto} \mathsf{W}[v_1, m_2']}$$

$$\frac{m_1 \mapsto m_1'}{\mathsf{RMW}[m_1, x{:}\tau.m_2, x{:}\tau.m_3] \overset{\emptyset}{\mapsto} \mathsf{RMW}[m_1', x{:}\tau.m_2, x{:}\tau.m_3]}$$

$$\frac{}{\mathsf{RMW}[\ell, x{:}\tau.m_b, x{:}\tau.m] \overset{v:\tau}{\longmapsto} \mathsf{RMWS}[\ell, v, [v/x]m_b, [v/x]m]}$$

$$\frac{m_b \mapsto m_b'}{\mathsf{RMWS}[\ell, v, m_b, m] \overset{\emptyset}{\mapsto} \mathsf{RMWS}[\ell, v, m_b', m]}$$

$$\frac{}{\mathsf{RMWS}[\ell, v_s, 0, m] \overset{\epsilon \# i = \mathsf{R}[l]\downarrow v_s}{\longmapsto} \mathsf{ret}\, v_s}$$

$$\frac{m \mapsto m'}{\mathsf{RMWS}[\ell, v, s(v_n), m] \overset{\emptyset}{\mapsto} \mathsf{RMWS}[\ell, v, s(v_n), m']}$$

$$\frac{}{\mathsf{RMWS}[\ell, v_s, s(v_n), v_m] \overset{\epsilon \# i = \mathsf{RW}[l, v_m]\downarrow v_s}{\longmapsto} \mathsf{ret}\, v_s}$$

$$\frac{}{\mathsf{fork}\, e \overset{\mathsf{fork}\, e\, \mathsf{as}\, p}{\longmapsto} \mathsf{ret}\ ()} \qquad \frac{}{\mathsf{alloc}\ v{:}\tau \overset{\mathsf{alloc}\ v:\tau\, \mathsf{as}\, \ell}{\longmapsto} \mathsf{ret}\, \ell}$$

$$\boxed{\xi \overset{\delta @ p}{\longmapsto} \xi'}$$

$$\frac{e \overset{\delta}{\mapsto} e'}{(\xi \parallel p : e) \overset{\delta @ p}{\longmapsto} (\xi \parallel p : e')} \qquad \frac{e \overset{\mathsf{fork}\, e''\, \mathsf{as}\, p'}{\longmapsto} e' \quad p' \text{ is fresh}}{(\xi \parallel p : e) \overset{\mathsf{fork}\, e''\, \mathsf{as}\, p' @ p}{\longmapsto} (\xi \parallel p : e' \parallel p' : e'')}$$

$$\frac{e \overset{\delta}{\mapsto} e' \quad \delta \text{ is not of the form } \mathsf{fork}\, e'' \,\mathsf{as}\, p'}{(\xi \parallel p : e) \overset{\delta@p}{\longmapsto} (\xi \parallel p : e')}$$

$\boxed{m \mapsto m'}$

$$\frac{m_1 \mapsto m_1'}{\mathsf{ifz}\, m_1 \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3 \mapsto \mathsf{ifz}\, m_1' \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3}$$

$$\frac{}{\mathsf{ifz}\, 0 \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3 \mapsto m_2} \qquad \frac{}{\mathsf{ifz}\, s(n) \,\mathsf{then}\, m_2 \,\mathsf{else}\, m_3 \mapsto m_3}$$

$$\frac{m_1 \mapsto m_1'}{m_1 m_2 \mapsto m_1' m_2} \qquad \frac{m_2 \to m_2'}{v_1 m_2 \mapsto v_1 m_2'}$$

$$\frac{}{(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.m)v \mapsto [(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.m), v/f, x]m}$$

A.4 The Store

$$\begin{array}{lrcl}
\text{events} & \theta & ::= & \mathsf{init}(i,p) \mid \mathsf{is}(i,\alpha) \mid \mathsf{spec}(i,v) \\
& & \mid & \mathsf{label}(\varphi,i) \mid \mathsf{edge}(b,T,T) \mid \mathsf{exec}(i) \mid \mathsf{rf}(i,i) \\
& & \mid & \mathsf{co}(i,i) \mid \mathsf{spo}(i,i) \mid \mathsf{fxo}(i,i) \\
\text{histories} & H & ::= & \epsilon \mid H, \theta
\end{array}$$

In the store transition rules, we write $H(\theta)$ to mean the event θ appears in H, where θ may contain wildcards ($*$) indicating parts of the event that don't matter. As usual, we write $\to^+$ and $\to^*$ for the transitive, and the reflexive, transitive closures of a relation $\to$. We write the composition of two relations as $\overset{x}{\to}\overset{y}{\to}$. We say that $\to$ is *acyclic* if $\neg\exists x.\, x \to^+ x$.

A.4.1 Synchronous store transitions

- *Tag declaration:* $\mathsf{tagdecl}_H(T) \stackrel{\text{def}}{=} H(\mathsf{edge}(*,T,*)) \vee H(\mathsf{edge}(*,*,T))$

$\boxed{H \overset{\delta@p}{\longmapsto} H'}$

$$\frac{}{H \overset{\emptyset@p}{\longmapsto} H}\ (\textsc{None}) \qquad \frac{}{H \overset{m:\tau@p}{\longmapsto} H}\ (\textsc{Type})$$

$$\frac{\neg H(\mathsf{init}(i,*))}{H \overset{\vec{\varphi}\#i=\alpha\downarrow v@p}{\longmapsto} H, \mathsf{init}(i,p), \mathsf{is}(i,\alpha), \mathsf{spec}(i,v), [\mathsf{label}(\varphi,i) \mid \varphi \in \vec{\varphi}]}\ (\textsc{Init})$$

$$\frac{T \neq T' \quad \neg\mathsf{tagdecl}_H(T) \quad \neg\mathsf{tagdecl}_H(T')}{H \overset{T \overset{b}{\Rightarrow} T' @ p}{\longmapsto} H, \mathsf{edge}(b, T, T')} \text{ (EDGE)}$$

$$\frac{\neg H(\mathsf{init}(i_1, *)) \quad \neg H(\mathsf{init}(i_2, *))}{H \overset{\mathsf{fork}\, e\, \mathsf{as}\, p_2 @ p_1}{\longmapsto} H, \mathsf{init}(i_1, p_1), \mathsf{is}(i_1, \mathsf{Push}), \mathsf{spec}(i_1, \ ()\ ), \mathsf{label}(\overset{\mathsf{vis}}{\triangleleft}, i_1), \\ \mathsf{init}(i_2, p_2), \mathsf{is}(i_2, \mathsf{Nop}), \mathsf{spec}(i_2, \ ()\ ), \mathsf{label}(\overset{\mathsf{exe}}{\triangleright}, i_2), \\ \mathsf{fxo}(i_1, i_2)} \text{ (FORK)}$$

$$\neg H(\mathsf{init}(i_w, *))$$
$$\neg H(\mathsf{init}(i_p, *))$$
$$H' = H, \mathsf{init}(i_w, p), \mathsf{is}(i_w, \mathsf{W_c}[\ell, v]), \mathsf{spec}(i_w, \ ()\ ), \\ \mathsf{init}(i_w, p), \mathsf{is}(i_w, \mathsf{Push}), \mathsf{spec}(i_p, \ ()\ ), \\ \mathsf{spo}(i_w, i_p)$$

$$\frac{\mathsf{executable}_{H'}(i_w) \qquad \mathsf{executable}_{H'; \mathsf{exec}(i_w)}(i_p) \\ \mathsf{acyclic}(\overset{\mathsf{co}}{\rightarrow}_{H', \mathsf{exec}(i_w), \mathsf{exec}(i_p)})}{H \overset{\mathsf{alloc}\ v:\tau\, \mathsf{as}\, \ell @ p}{\longmapsto} H', \mathsf{exec}(i_w), \mathsf{exec}(i_p)} \text{ (ALLOC)}$$

A.4.2 Store orderings

- Identifiers are in *program order* if they are initiated in order and on the same thread:

$$i \overset{\mathsf{po}}{\rightarrow}_H i' \overset{\text{def}}{=} \exists H_1, H_2, H_3, p.\ H = H_1, \mathsf{init}(i, p), H_2, \mathsf{init}(i', p), H_3$$

- An identifier is marked as executed by either an exec or an rf event:
  $\mathsf{exec}_H(i) \overset{\text{def}}{=} H(\mathsf{exec}(i)) \vee H(\mathsf{rf}(*, i))$.
- *Trace order* is the order in which identifiers were actually executed:
  $i \overset{\mathsf{to}}{\rightarrow}_H i' \overset{\text{def}}{=} \exists H_1, H_2.\ H = H_1, H_2 \wedge \mathsf{exec}_{H_1}(i) \wedge \neg\mathsf{exec}_{H_1}(i')$.
  Note that this definition makes executed identifiers trace-order-earlier than non-yet-executed identifiers.
- *Specified order,* which realizes the tagging discipline, is defined by the rules:

$$\frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad H(\mathsf{label}(T, i)) \quad H(\mathsf{edge}(b, T, T')) \quad H(\mathsf{label}(T', i'))}{i \overset{b}{\Rightarrow}_H i'}$$

$$\frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad H(\mathsf{label}(\overset{b}{\triangleleft}, i'))}{i \overset{b}{\Rightarrow}_H i'} \qquad \frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad H(\mathsf{label}(\overset{b}{\triangleright}, i))}{i \overset{b}{\Rightarrow}_H i'}$$

- The key notion of *execution order* is defined in terms of specified order: $i \xrightarrow{\mathsf{xo}}_H i' \stackrel{\text{def}}{=} \exists b.\, i \overset{b}{\Rightarrow}_H i'$.
- The action types stored in the history allow us to query to type of an action: $\mathsf{is\_type}_H(i,t) = H(\mathsf{is}(i,\mathsf{R}_t[\ell])) \vee H(\mathsf{is}(i,\mathsf{RW}_t[\ell,*])) \vee H(\mathsf{is}(i,\mathsf{W}_t[\ell,*]))$.
- A read action is either a read or a read-write: $\mathsf{reads}_H(i,\ell) = H(\mathsf{is}(i,\mathsf{R}[\ell])) \vee H(\mathsf{is}(i,\mathsf{RW}[\ell,*]))$.
- Similarly, a write action is either a write or a read-write: $\mathsf{writes}_H(i,\ell,v) = H(\mathsf{is}(i,\mathsf{W}[\ell,v])) \vee H(\mathsf{is}(i,\mathsf{RW}[\ell,v]))$.
- An action i accesses some location l if it either reads or writes to l: $\mathsf{accesses}_H(i,l) = \mathsf{reads}_H(i,l) \vee \mathsf{writes}_H(i,l,*)$
- Two accesses access the same location if they do: $\mathsf{sameloc}_H(i,i') = \exists l.\mathsf{accesses}_H(i,l) \wedge \mathsf{accesses}_H(i',l)$
- *Reads-from:* $i \xrightarrow{\mathsf{rf}}_H i' \stackrel{\text{def}}{=} H(\mathsf{rf}(i,i'))$.
- *Fork-execution-order:* $i \xrightarrow{\mathsf{fxo}}_H i' \stackrel{\text{def}}{=} H(\mathsf{fxo}(i,i'))$.
- *Spontaneous-push-order:* $i \xrightarrow{\mathsf{spo}}_H i' \stackrel{\text{def}}{=} H(\mathsf{spo}(i,i'))$.
- An identifier is *executable* if (1) it has not been executed, (2) all its execution-order predecessors have been, (3) all of its spontaneous-push-order predecessors are executed, (4) all of its push-edge-order predecessors are spontaneous-push-ordered before a push that has been executed, and (5) all of its fork-order predecessors are executed:

$$\begin{aligned}\mathsf{executable}_H(i) \stackrel{\text{def}}{=} {} & \neg\mathsf{exec}_H(i) \\ & \wedge (\forall i'.\, i' \xrightarrow{\mathsf{xo}}_H i \supset \mathsf{exec}_H(i')) \\ & \wedge (\forall i'.\, i' \xrightarrow{\mathsf{spo}}_H i \supset \mathsf{exec}_H(i')) \\ & \wedge (\forall i'.\, i' \overset{\mathsf{push}}{\Rightarrow}_H i \supset \exists i''.\, H(\mathsf{is}(i'',\mathsf{Push}) \wedge \mathsf{exec}_H(i'')) \wedge i' \xrightarrow{\mathsf{spo}}_H i'') \\ & \wedge (\forall i'.\, i' \xrightarrow{\mathsf{fxo}}_H i \supset \mathsf{exec}_H(i'))\end{aligned}$$

- Identifiers i and i' are *push-ordered* if i is a push and is trace-order-earlier than i': $i \xrightarrow{\pi\mathsf{o}}_H i' \stackrel{\text{def}}{=} H(\mathsf{is}(i,\mathsf{Push})) \wedge i \xrightarrow{\mathsf{to}}_H i'$
  Since pushes are globally visible as soon as they execute, this means that i should be visible to i'.
- An identifier is *concurrent* if it is a concurrent or an SC action (that is, not a plain one): $\mathsf{concurrent}_H(i) = \mathsf{is\_type}_H(i,\mathsf{c}) \vee \mathsf{is\_type}_H(i,\mathsf{sc})$
- *Atomically-reads-from* is reads-from restricted to concurrent identifiers: $i \xrightarrow{\mathsf{arf}}_H i' \stackrel{\text{def}}{=} H(\mathsf{rf}(i,i')) \wedge \mathsf{concurrent}_H(i) \wedge \mathsf{concurrent}_H(i')$
- The key notion of *visibility order* is defined as the union of specified visibility order, atomically-reads-from, and push order: $i \xrightarrow{\mathsf{vo}}_H i' \stackrel{\text{def}}{=} i \overset{\mathsf{vis}}{\Rightarrow}_H i' \vee i \xrightarrow{\mathsf{arf}}_H i' \vee i \xrightarrow{\pi\mathsf{o}}_H i' \vee i \xrightarrow{\mathsf{spo}}_H i'$.
- *Built-in sequential consistency:* $i \xrightarrow{\mathsf{bsc}}_H i' \stackrel{\text{def}}{=} \mathsf{is\_type}_H(i,\mathsf{sc}) \wedge \mathsf{is\_type}_H(i',\mathsf{sc}) \wedge i \xrightarrow{\mathsf{to}}_H i'$

- *Priority*: Priority is defined as the transitive closure of per-location program order and visible-to:

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad \mathsf{sameloc}_H(i,i')}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-PO)} \qquad \frac{i \xrightarrow{\mathsf{vt}}_H i' \quad \mathsf{sameloc}_H(i,i')}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-VT)}$$

$$\frac{i \xrightarrow{\mathsf{bsc}}_H i' \quad \mathsf{sameloc}_H(i,i')}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-SC)}$$

$$\frac{i \xrightarrow{\mathsf{pri}}_H i'' \quad i'' \xrightarrow{\mathsf{pri}}_H i'}{i \xrightarrow{\mathsf{pri}}_H i'} \text{ (PRI-TRANS)}$$

When $i \xrightarrow{\mathsf{pri}} i'$, we say that i is *prior to* i' or that i' "sees" i.

Coherence Order

$$\frac{i \xrightarrow{\mathsf{pri}}_H i_r \quad i' \xrightarrow{\mathsf{rf}}_H i_r \quad \mathsf{writes}_H(i,l,v) \quad i \neq i'}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-READ)}$$

$$\frac{i \xrightarrow{\mathsf{pri}}_H i' \quad \mathsf{writes}_H(i,l,v) \quad \mathsf{writes}_H(i',l,v')}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-WRITE)}$$

$$\frac{i_w \xrightarrow{\mathsf{rf}}_H i \quad H(\mathsf{is}(i,\mathsf{RW}[*,*])) \quad i_w \xrightarrow{\mathsf{co}}_H i' \quad i \neq i'}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-RW)}$$

$$\frac{H(\mathsf{co}(i,i'))}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-FIAT)}$$

A.4.3 Asynchronous store transitions

$\boxed{H \rightsquigarrow H'}$

$$\frac{\begin{array}{l} H(\mathsf{spec}(i,v)) \quad \mathsf{reads}_H(i,\ell) \\ \mathsf{writes}_H(i_w,\ell,v) \quad \mathsf{exec}_H(i_w) \\ i_w \xrightarrow{\mathsf{pri}}_{H;\mathsf{rf}(i_w,i)} i \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H;\mathsf{rf}(i_w,i)}) \end{array}}{H \rightsquigarrow H, \mathsf{rf}(i_w,i)} \text{ (READ)}$$

$$\frac{\begin{array}{c} H(\mathsf{is}(i,\alpha)) \quad H(\mathsf{spec}(i,\ ()\ )) \\ \alpha = \mathsf{W}[\ell,v] \vee \alpha = \mathsf{Push} \vee \alpha = \mathsf{Nop} \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\overset{\mathsf{co}}{\rightarrow}_{H;\mathsf{exec}(i)}) \end{array}}{H \rightsquigarrow H, \mathsf{exec}(i)}\ (\textsc{Nonread})$$

$$\frac{\begin{array}{c} S \text{ is a set of actions} \\ \neg H(\mathsf{init}(i,*)) \\ \forall i' \in S.\, H(\mathsf{init}(i',p)) \end{array}}{H \rightsquigarrow H, \mathsf{init}(i,p), \mathsf{is}(i,\mathsf{Push}), \mathsf{spec}(i,\ ()\ ),\ [\mathsf{spo}(i',i) \mid i' \in S]}\ (\textsc{Push-Init})$$

A.5 Trace coherence

$\boxed{H\ \mathsf{trco}}$

$$\frac{}{\epsilon\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad \neg H(\mathsf{init}(i,*))}{H, \mathsf{init}(i,p)\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad H = H', \mathsf{init}(i,p)}{H, \mathsf{is}(i,\alpha)\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad H = H', \mathsf{is}(i,\alpha)}{H, \mathsf{spec}(i,v)\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad H(\mathsf{init}(i,*)) \quad H(\mathsf{init}(i',*)) \quad H(\mathsf{is}(i,\mathsf{Push})) \quad H(\mathsf{is}(i',\mathsf{Nop}))}{H, \mathsf{fxo}(i,i')\ \mathsf{trco}}$$

$$\frac{\begin{array}{c} H\ \mathsf{trco} \\ H(\mathsf{spec}(i_r,v)) \quad \mathsf{reads}_H(i_r,\ell) \\ \mathsf{writes}_H(i_w,\ell,v) \quad \mathsf{exec}_H(i_w) \\ \mathsf{executable}_H(i_r) \end{array}}{H, \mathsf{rf}(i_w,i_r)\ \mathsf{trco}} \qquad \frac{\begin{array}{c} H\ \mathsf{trco} \\ H(\mathsf{is}(i,\alpha)) \quad H(\mathsf{spec}(i,\ ()\ )) \\ \alpha = \mathsf{W}[\ell,v] \vee \alpha = \mathsf{Push} \vee \alpha = \mathsf{Nop} \\ \mathsf{executable}_H(i) \end{array}}{H, \mathsf{exec}(i)\ \mathsf{trco}}$$

$$\frac{\begin{array}{c} H\ \mathsf{trco} \\ H(\mathsf{is}(i,\mathsf{Push})) \quad H(\mathsf{init}(i,p)) \\ (\exists H'.\, H = H', \mathsf{spec}(i,\ ()\ )) \vee (\exists H', i'.\, H = H', \mathsf{spo}(i'',i)) \\ H(\mathsf{init}(i',p)) \end{array}}{H, \mathsf{spo}(i',i)\ \mathsf{trco}}$$

$$\frac{H\ \mathsf{trco} \quad T \neq T' \quad \neg\mathsf{tagdecl}_H(T) \quad \neg\mathsf{tagdecl}_H(T')}{H, \mathsf{edge}(b, T, T')\ \mathsf{trco}}$$

$$\frac{\begin{array}{l} H\ \mathsf{trco} \\ (\exists H', v.\, H = H', \mathsf{spec}(i, v)) \vee (\exists H', \varphi'.\, H = H', \mathsf{label}(\varphi', i)) \\ \mathsf{labeldecl}_H(\varphi) \end{array}}{H, \mathsf{label}(\varphi, i)\ \mathsf{trco}}$$

$$\frac{\mathsf{tagdecl}_H(T)}{\mathsf{labeldecl}_H(T)} \qquad \frac{}{\mathsf{labeldecl}_H(\overset{b}{\triangleleft})} \qquad \frac{}{\mathsf{labeldecl}_H(\overset{b}{\triangleright})}$$

$$\frac{H\ \mathsf{trco} \quad \mathsf{writes}_H(i, \ell, v) \quad \mathsf{writes}_H(i', \ell, v')}{H, \mathsf{co}(i, i')\ \mathsf{trco}}$$

A.6 Store static semantics

$$\text{signature} \quad \Sigma \quad ::= \quad (\Upsilon, \Phi)$$

$\boxed{\vdash \Upsilon\ \mathsf{ok} \quad \vdash H : \Upsilon}$

$$\frac{}{\vdash \epsilon\ \mathsf{ok}} \qquad \frac{\vdash \Upsilon\ \mathsf{ok} \quad T \notin \Upsilon}{\vdash \Upsilon, T\ \mathsf{ok}}$$

$$\frac{\vdash \Upsilon\ \mathsf{ok} \quad \forall T.\, T \in \Upsilon \Leftrightarrow \mathsf{tagdecl}_H(T)}{\vdash H : \Upsilon}$$

$\boxed{\vdash \Phi\ \mathsf{ok} \quad \Upsilon; \Phi_0 \vdash H : \Phi}$

$$\frac{}{\vdash \epsilon\ \mathsf{ok}} \qquad \frac{\vdash \Phi\ \mathsf{ok} \quad \ell \notin \mathsf{Dom}(\Phi)}{\vdash \Phi, \ell{:}\tau\ \mathsf{ok}}$$

$$\frac{\begin{array}{l} \vdash \Phi\ \mathsf{ok} \\ \forall(\ell{:}\tau){\in}\Phi.\, \mathsf{initialized}_H(\ell) \\ \forall(\ell{:}\tau){\in}\Phi.\, \forall i, v.\, \mathsf{writes}_H(i, \ell, v) \supset \Upsilon; \Phi_0; \epsilon \vdash v : \tau \\ \forall i, \ell, v.\, \mathsf{writes}_H(i, \ell, v) \supset \ell \in \mathsf{Dom}(\Phi) \end{array}}{\Upsilon; \Phi_0 \vdash H : \Phi}$$

$$\begin{array}{l} \mathsf{initialized}_H(\ell) \stackrel{\text{def}}{=} \\ \exists i_w, i_p, v.\, \mathsf{writes}_H(i_w, \ell, v) \wedge H(\mathsf{is}(i_p, \mathsf{Push})) \\ \qquad \wedge\, i_w \overset{\mathsf{vo}}{\rightarrow}{}^{+}_H\ i_p \wedge \mathsf{exec}_H(i_p) \\ \qquad \wedge\, \forall i_r.\, \mathsf{reads}_H(i_r, \ell) \supset i_p \overset{\mathsf{to}}{\rightarrow}_H\ i_r \end{array}$$

$\boxed{\vdash H : \Sigma}$

$$\frac{H\ \mathsf{trco} \quad \mathsf{acyclic}(\overset{\mathsf{co}}{\rightarrow}_H) \qquad \vdash H : \Upsilon \quad \Upsilon; \Phi \vdash H : \Phi}{\vdash H : (\Upsilon, \Phi)}$$

A.7 Signature dynamic semantics

$\boxed{\Sigma \overset{\delta@p}{\longmapsto} \Sigma'}$

$$\frac{\Upsilon, \Phi, \epsilon \vdash m : \tau}{(\Upsilon, \Phi) \overset{m:\tau@p}{\longmapsto} (\Upsilon, \Phi)} \qquad \frac{\forall \varphi \in \vec{\varphi}.\ \Upsilon; \epsilon \vdash \varphi : \mathsf{label} \quad \Upsilon, \Phi, \epsilon \vdash \alpha : \tau \quad \Upsilon, \Phi, \epsilon \vdash v : \tau}{(\Upsilon, \Phi) \overset{\vec{\varphi} \# i = \alpha \downarrow v@p}{\longmapsto} (\Upsilon, \Phi)}$$

$$\frac{}{\Sigma \overset{\emptyset@p}{\longmapsto} \Sigma} \qquad \frac{}{\Sigma \overset{\mathsf{fork}\, e\, \mathsf{as}\, p'@p}{\longmapsto} \Sigma}$$

$$\frac{T \neq T' \quad T, T' \notin \Upsilon}{(\Upsilon, \Phi) \overset{T \overset{b}{\Rightarrow} T'@p}{\longmapsto} ((\Upsilon, T, T'), \Phi)}$$

$$\frac{\Upsilon; \Phi; \epsilon \vdash v : \tau \quad \ell \notin \mathsf{Dom}(\Phi)}{(\Upsilon, \Phi) \overset{\mathsf{alloc}\ v:\tau \mathsf{as}\ \ell@p}{\longmapsto} (\Upsilon, (\Phi, \ell{:}\tau))}$$

A.8 Top-level semantics

$$\text{state} \quad s \quad ::= \quad (\Sigma, H, \xi)$$

$\boxed{\vdash s\ \mathsf{ok}}$

$$\frac{\vdash H : (\Upsilon, \Phi) \quad \Upsilon, \Phi \vdash \xi\ \mathsf{ok}}{\vdash ((\Upsilon, \Phi), H, \xi)\ \mathsf{ok}}$$

$\boxed{s \mapsto s'}$

$$\frac{\Sigma \overset{\delta@p}{\longmapsto} \Sigma' \quad H \overset{\delta@p}{\longmapsto} H' \quad \xi \overset{\delta@p}{\longmapsto} \xi'}{(\Sigma, H, \xi) \mapsto (\Sigma', H', \xi')} \qquad \frac{H \rightsquigarrow H'}{(\Sigma, H, \xi) \mapsto (\Sigma, H', \xi)}$$

A.9 Side conditions, etc

A.9.1 Mootness

$\boxed{s \text{ complete}}$

$$\frac{\forall i, p.\, H(\mathsf{init}(i,p)) \supset \mathsf{exec}_H(i)}{(\Sigma, H, \xi)\ \mathsf{complete}}$$

$\boxed{s \text{ moot}}$

$$\frac{\nexists s'.\, s \mapsto^* s' \wedge s' \text{ complete}}{s\ \mathsf{moot}}$$

Some important properties of mootness include:

1. If s moot and $s \mapsto^* s'$, then s' moot.
2. If $(\Sigma, H, \xi \parallel p : e)$ moot, then (Σ, H, ξ) moot.
3. If (Σ, H, ξ) cannot take a step, and there are unexecuted actions in H, then (Σ, H, ξ) is moot.
4. If every reduct of s is moot, then s moot.

A.9.2 Data races

- $\mathsf{samethread}_H(i, i') = \exists p.H(\mathsf{init}(i,p))) \wedge H(\mathsf{init}(i', p))$
- $\mathsf{conflict}_H(i, i') = \exists l, v.\mathsf{sameloc}_H(i, i') \wedge (\mathsf{writes}_H(i, l, v) \vee \mathsf{writes}_H(i', l, v))$

$$\frac{\mathsf{conflict}_H(i,i') \quad \neg\mathsf{samethread}_H(i,i') \quad \neg i \overset{\mathsf{vt}}{\rightarrow}_H i' \quad \neg i' \overset{\mathsf{vt}}{\rightarrow}_H i \quad \mathsf{is\_type}_H(i, \mathsf{p}) \vee \mathsf{is\_type}_H(i', \mathsf{p})}{(\Sigma, H, \xi)\ \mathsf{races}}$$

Then we say that if there exists *any* non-moot execution that contains a data race, the program is undefined:

$$\frac{s \mapsto^* s' \quad s' \text{ races} \quad \neg s' \text{ moot}}{s\ \mathsf{catches{-}fire}}$$

A.9.3 Liveness

1. All executable actions will eventually be executed.
2. Writes will eventually become visible. That is, a thread will not indefinitely read from a write when there is another write coherence-after it.

A.9.4 Thin-air

- "Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation." [28].

Appendix B

Some proofs

B.1 Sequential consistency for SC operations

This proof is very closely structured after the proofs in the original RMC paper [18].

A *from-read* edge between i and i' means that i reads from a write that is coherence-earlier than i'. *Sequentially consistent order* [3] is the union of $\overset{\mathsf{bsc}}{\to}$, $\overset{\mathsf{co}}{\to}$, $\overset{\mathsf{rf}}{\to}$, and $\overset{\mathsf{fr}}{\to}$:

$$\begin{array}{lcl} i \overset{\mathsf{bsc}}{\to}_H i' & \overset{\text{def}}{=} & \mathsf{is\_type}_H(i, \mathsf{sc}) \wedge \mathsf{is\_type}_H(i', \mathsf{sc}) \wedge i \overset{\mathsf{to}}{\to}_H i' \\ i \overset{\mathsf{fr}}{\to}_H i' & \overset{\text{def}}{=} & \exists i_w.\, i_w \overset{\mathsf{rf}}{\to}_H i \wedge i_w \overset{\mathsf{co}}{\to}^+_H i' \\ i \overset{\mathsf{sc}}{\to}_H i' & \overset{\text{def}}{=} & i \overset{\mathsf{bsc}}{\to}_H i' \vee i \overset{\mathsf{co}}{\to}_H i' \vee i \overset{\mathsf{rf}}{\to}_H i' \vee i \overset{\mathsf{fr}}{\to}_H i' \end{array}$$

We also define *communication order* [3] as sequentially consistent order without $\overset{\mathsf{bsc}}{\to}$:

$$i \overset{\mathsf{com}}{\to}_H i' \overset{\text{def}}{=} i \overset{\mathsf{co}}{\to}_H i' \vee i \overset{\mathsf{rf}}{\to}_H i' \vee i \overset{\mathsf{fr}}{\to}_H i'$$

An important property of communication order is that it relates only accesses to the same location, and it agrees with coherence order on writes. Since it always relates at least one write, this means it is acyclic iff coherence is (Consider $i \overset{\mathsf{com}}{\to}^+_H i$. If i is a write, then $i \overset{\mathsf{co}}{\to}^+_H i$. If i is a read, then $i \overset{\mathsf{com}}{\to}^*_H i_w \overset{\mathsf{rf}}{\to}_H i$, so $i_w \overset{\mathsf{co}}{\to}^+_H i_w$.).

Lemma 9 (Main Lemma)**.** *Suppose H is trace coherent, complete (that is, every identifier in H is executed), and coherence acyclic (that is,* $\mathsf{acyclic}(\overset{\mathsf{co}}{\to}_H)$*). If $i_1 \overset{\mathsf{com}}{\to}^+_H i_2$, where i_1 and i_2 are* sc *operations, then $i_1 \overset{\mathsf{to}}{\to}_H i_2$.*

Proof: First note that i_1 and i_2 must both be reads or writes (sc only can apply to those).

Since H is complete, i_1 and i_2 are executed. Thus we have either $i_1 \overset{\mathsf{to}}{\to}_H i_2$, $i_1 = i_2$, or $i_2 \overset{\mathsf{to}}{\to}_H i_1$.

If $i_1 = i_2$ and i_1 is a write, then we have $i_1 \overset{\mathsf{co}}{\to}^+_H i_1$, which is a contradiction. If i_1 is a read, then we have $i_1 \overset{\mathsf{fr}}{\to}_H i_w \overset{\mathsf{co}}{\to}^+_H i'_w \overset{\mathsf{rf}}{\to}_H i_1$. But $i_1 \overset{\mathsf{fr}}{\to}_H i_w$ implies that $i'_w \overset{\mathsf{co}}{\to}^+_H i_w$, which is a contradiction. (None of this even depends on being SC.)

If $i_2 \overset{\mathsf{to}}{\to}_H i_1$, i_1 and i_2 being SC operations implies $i_2 \overset{\mathsf{bsc}}{\to}_H i_1$. Since they are on the same location, this implies $i_2 \overset{\mathsf{pri}}{\to}_H i_1$. Then:

- First, note that we have some i_w such that $i_1 \overset{\mathsf{com}}{\to}{}^*_H\ i_w$ and $i_w \overset{\mathsf{pri}}{\to}_H i_1$. If i_2 is a write then this is trivially $i_w = i_2$. If i_2 is a read, then we have $i_1 \overset{\mathsf{com}}{\to}{}^*_H\ i_w \overset{\mathsf{rf}}{\to}_H i_2$. We obtain $i_w \overset{\mathsf{pri}}{\to}_H i_1$ because $i_w \overset{\mathsf{vt}}{\to}_H i_2$ and therefore $i_w \overset{\mathsf{pri}}{\to}_H i_2$.
- If i_1 is a write, then this gives us $i_w \overset{\mathsf{co}}{\to}_H i_1$ and thus $i_w \overset{\mathsf{com}}{\to}{}^+_H\ i_w$, which is a contradiction.
- If i_1 is a read, then we have some i such that $i \overset{\mathsf{rf}}{\to}_H i_1$. Since $i_w \overset{\mathsf{pri}}{\to}_H i_1$, this means that $i_w \overset{\mathsf{co}}{\to}_H i$. Then, since $i_w \overset{\mathsf{co}}{\to}_H i$, $i \overset{\mathsf{rf}}{\to}_H i_1$, and $i_1 \overset{\mathsf{com}}{\to}{}^*_H\ i_w$, we obtain $i_w \overset{\mathsf{com}}{\to}{}^+_H\ i_w$, which is a contradiction.

□

Corollary 10. *Suppose H is trace coherent, complete, and coherence acyclic. If $i_1 \overset{\mathsf{sc}}{\to}{}^+_H\ i_2$, where i_1 and i_2 are* sc *operations, then $i_1 \overset{\mathsf{to}}{\to}_H i_2$.*

Proof: By induction on the number of $\overset{\mathsf{bsc}}{\to}$ steps in $i_1 \overset{\mathsf{sc}}{\to}{}^+_H\ i_2$. □

Theorem 11. *Suppose H is trace coherent, complete, and coherence acyclic. Then $\overset{\mathsf{sc}}{\to}_H$ is acyclic.*

Proof: Suppose $i \overset{\mathsf{sc}}{\to}{}^+_H\ i$. There must be at least one $\overset{\mathsf{bsc}}{\to}$ edge in the cycle. Otherwise we have $i \overset{\mathsf{com}}{\to}{}^+_H\ i$, which is a contradiction.

Thus suppose that $i_1 \overset{\mathsf{bsc}}{\to}_H i_2 \overset{\mathsf{sc}}{\to}{}^*_H\ i_1$. Expanding $\overset{\mathsf{bsc}}{\to}$ we obtain $i_1 \overset{\mathsf{to}}{\to}_H i_2 \overset{\mathsf{sc}}{\to}{}^*_H\ i_1$ with i_1 and i_2 as sc actions. By Corollary 10, $i_2 \overset{\mathsf{to}}{\to}_H i_1$, which is a contradiction. □

The rest of the proof proceeds basically exactly the same as the paper proof for using push edges, with a minor tweak to allow us to make a statement about sc operations that are *not* constrained by edges.

Definition 12. A history H is *consistent* if there exists H' containing no is events, such that H, H' is trace coherent, complete, and coherence acyclic.

Definition 13. Suppose R is a binary relation over identifiers and S is a set of identifiers. Then R is a *weakly sequentially consistent ordering for* S if (1) R is a total order, (2) R respects *trace* order for elements of S, and (3) every read in the domain of R is satisfied by the most R-recent write to its location.

Corollary 14 (Weak sequential consistency)**.** *Suppose H is consistent. Suppose further that S is a set of identifiers such that for all $i \in S$,* $\mathsf{is\_type}_H(i, \mathsf{sc})$*. Then there exists a weakly sequentially consistent ordering for S.*

Proof: Let H, H' be trace coherent, complete, and coherence acyclic. Let H'' extend H, H' by making enough coherence commitments to totally order all writes to the same location. (The CO-RW rule makes this tricky, since we must join read-write chains only at the ends, but it's not hard to show it can be done.) Then H'' too is trace coherent, complete, and coherence acyclic.

By Theorem 11, $\overset{\mathsf{sc}}{\to}_{H''}$ is acyclic, so there exists a total order containing it. Let R be such an order. Following Alglave [3], we show that R is a weakly sequentially consistent order for S:

- Suppose $i, i' \in S$ and $i \overset{\mathsf{to}}{\to}_H i'$. This implies $i \overset{\mathsf{bsc}}{\to}_H i'$. Since H'' extends H, we have $i \overset{\mathsf{bsc}}{\to}_{H''} i'$, so $i \overset{\mathsf{sc}}{\to}_{H''} i'$, so $i\ R\ i'$.
- Suppose $i_w \overset{\mathsf{rf}}{\to}_H i_r$. Then $i_w \overset{\mathsf{rf}}{\to}_{H''} i_r$, so $i_w \overset{\mathsf{sc}}{\to}_{H''} i_r$, so $i_w\ R\ i_r$. Let ℓ be the location that i_w writes and i_r reads. It remains to show there is no write i'_w to ℓ such that $i_w\ R\ i'_w\ R\ i_r$.

Suppose i'_w is a write to ℓ and $i_w \neq i'_w$. Either $i'_w \xrightarrow{\mathsf{co}}_{H''} i_w$ or $i_w \xrightarrow{\mathsf{co}}_{H''} i'_w$. If $i'_w \xrightarrow{\mathsf{co}}_{H''} i_w$ then $i'_w \; R \; i_w$. If $i_w \xrightarrow{\mathsf{co}}_{H''} i'_w$, then $i_r \xrightarrow{\mathsf{fr}}_{H''} i'_w$, so $i_r \; R \; i'_w$.

□

Definition 15. Suppose R is a binary relation over identifiers and S is a set of identifiers. Then R is a *sequentially consistent ordering for* S if (1) R is a total order, (2) R respects *program* order for elements of S, and (3) every read in the domain of R is satisfied by the most R-recent write to its location[1].

Corollary 16 (Sequential consistency)**.** *Suppose H is consistent. Suppose further that S is a set of identifiers such that for all $i \in S$, $\mathsf{is\_type}_H(i, \mathsf{sc})$ and $H(\mathsf{label}(\overset{\mathsf{exe}}{\triangleright}, i))$. Then there exists a sequentially consistent ordering for S.*

Proof: Note that for identifiers in S, since they are tagged with execution post-edges, program order implies trace order. Thus the weakly sequentially consistent ordering promised by Corollary 14 is also a sequentially consistent ordering. □

Observe that the sequentially consistent order includes *all* of H's writes, not only those belonging to S. Thus, writes not belonging to S are adopted into the sequentially consistent order. This means that the members of S have a consistent view of the order in which *all* writes took place, not just the writes belonging to S. However, for non-members that order might not agree with program order. (The order contains non-S reads too, but for purposes of reasoning about S we can ignore them. Only the writes might satisfy a read in S.)

[1] This is the same as weak sequential consistency except that it must respect program order instead of trace order.

Bibliography

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. In *CAV*, 2014. 4.4.3, 7.3

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 2014. 1.2.1

[3] Jave Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris VII, November 2010. B.1, B.1

[4] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. Number ARM DDI 0487B.a. March 2017. 4.3, 4.5, 4.5.1

[5] David Aspinall and Jaroslav Ševčík. Java memory model examples: Good, bad and ugly. In *VAMP 2007*, 2007. 1.3.1

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Thirty-Eighth ACM Symposium on Principles of Programming Languages*, January 2011. 7.4

[7] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Thirty-Ninth ACM Symposium on Principles of Programming Languages*, Philadelphia, Pennsylvania, January 2012. 4.1

[8] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015. 4.4.3, 7.3

[9] OpenMP Architecture Review Board. Openmp application programming interface. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`, November 2015. 7.3

[10] Hans-J. Boehm. Threads cannot be implemented as a library. In *2005 SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005. 1.3

[11] Hans-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. ACM, 2012. 5.6

[12] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *2008 SIGPLAN Conference on Programming Language Design and Implementation*,

Tucson, Arizona, June 2008. 1.3, 3.3.2

[13] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, 2014. 3.2.4

[14] 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*, 2013. 4.4.3, 7.3

[15] Sam Bowen and Josh Zimmerman. Lockless data structures on the rmc memory model. `http://www.contrib.andrew.cmu.edu/~sgbowen/15418/`, 2015. 5.8

[16] Jonathan Corbet. Mcs locks and qspinlocks. `https://lwn.net/Articles/590243/`, 2014. 5.5

[17] Marie-Christine Costa, Lucas Létocart, and Frédéric Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 2005. 4.4.3

[18] Karl Crary and Michael J. Sullivan. A calculus for relaxed memory. In *Forty-Second ACM Symposium on Principles of Programming Languages*, Mumbai, India, January 2015. 1, 2, 2.3.2, 3.1, 3.3.2, B.1

[19] Karl Crary, Joseph Tassarotti, and Michael J. Sullivan. Relaxed memory calculus 2.0, draft. `https://www.cs.cmu.edu/~jtassaro/papers/rmc2.pdf`, 2017. 3.1, 3.4

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg, 2008. 4.4.3

[21] Will Deacon. The armv8 application level memory model. From `herdtools7`, `https://github.com/herd/herdtools7/blob/7.47/herd/libdir/aarch64.cat`, 2017. 4.5.1

[22] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, Eindhoven University of Technology, 1965. 1.2.1

[23] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf`. 5.3.1, 6.1

[24] Stephen Hemminger. fast reader/writer lock for gettimeofday 2.5.30. Linux kernel mailing list August 12, 2002, `https://lwn.net/Articles/7388/`, 2002. 5.6

[25] David Howells and Paul E. McKenney. Linux kernel memory barriers. `https://www.kernel.org/doc/Documentation/memory-barriers.txt`, 2006. 1.3

[26] David Howells and Paul E. McKenney. Circular buffers. `https://www.kernel.org/doc/Documentation/circular-buffers.txt`, 2010. 2.2.1

[27] Intel Corporation. *Intel® IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture*. Number 245318-002. July 2000. 3.2.2

[28] ISO/IEC 14882:2014. Programming languages – c++. `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=`

64029 or a good approximation at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf, 2014. 2.4.1, 3.2.4, A.9.4

[29] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Forty-Fourth ACM Symposium on Principles of Programming Languages*, Paris, France, January 2017. 2.4.1, 3.2.4, 3.2.4

[30] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11, 2017. 4.7

[31] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979. 1, 1.1

[32] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Thirty-Second ACM Symposium on Principles of Programming Languages*, Long Beach, California, January 2005. 1.3, 3.2.4

[33] Paul E. McKenney. Proper care and feeding of return values from rcu_dereference(). https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt, 2014. 1.3, 5.7.2, 6.1

[34] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency. In *Parallel and Distributed Computing and Systems*, 1998. 2.2.2, 5.7

[35] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, Oliver Giroux, Lawrence Crowl, JF Bastien, and Michael Wong. Marking memory_order_consume dependency chains. C++ standards committee paper WG21/P0462R1,http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0462r1.pdf, 2017. 1.3.2, 7.3

[36] Paul E. McKenney, Michael Wong, Hans Boehm, Jens Maurer, Jeffrey Yasskin, and JF Bastien. Proposal for new memory_order_consume definition. C++ standards committee paper WG21/P0190R4,http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0190r4.pdf, 2017. 1.3.2, 7.3

[37] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 5.5

[38] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996. 5.4, 6.1

[39] Peizhao Ou and Brian Demsky. Automo: Automatic inference of memory order parameters for c/c++11. In *2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2015. 7.3

[40] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *2011 SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, California, June 2011. 1.2.1, 3.1.4, 3.1.4, 4.3

[41] Jaroslav Sevcik and Peter Sewell. C/c++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html, 2016. 4.5.2

[42] 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. *Communications of the ACM*, 53(7), July 2010. 1.2.1, 4.2

[43] Michael J. Sullivan. rmc-compiler. `https://github.com/msullivan/rmc-compiler`, 2015. 4

[44] Ian Lance Taylor. Single threaded memory model. `http://www.airs.com/blog/archives/79`, 2007. 1.2.2

[45] R. Kent Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986. 5.2, 6.1

[46] Aaron Turon. Crossbeam: support for concurrent and parallel programming. `https://github.com/aturon/crossbeam`, 2015. 5.3.1, 5.4

[47] 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 *Forty-Second ACM Symposium on Principles of Programming Languages*, POPL '15, 2015. 3.2.4

[48] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, 2008. 3.2.4

[49] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. 3.3.1